Sub Code & Name: (15A04306) & DIGITAL LOGIC DESIGN	
Branch: CSE &IT	Semester: II – I

UNIT I
NUMBER SYSTEM & BOOLEAN ALGEBRA

A digital system can understand positional number system only where there are a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

A value of each digit in a number can be determined using

- The digit
- The position of the digit in the number
- The base of the number system (where base is defined as the total number of digits available in the number system).

Decimal Number System

The number system that we use in our day-to-day life is the decimal number system. Decimal number system has base 10 as it uses 10 digits from 0 to 9. In decimal number system, the successive positions to the left of the decimal point represents units, tens, hundreds, thousands and so on.

Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position, and its value can be written as

$(1\times1000) + (2\times100) + (3\times10) + (4\times1)$

$(1\times10^3) + (2\times10^2) + (3\times10^1) + (4\times10^0)$

$1000 + 200 + 30 + 1$

1234

As a computer programmer or an IT professional, you should understand the following number systems which are frequently used in computers.

S.N.	Number System & Description
1	**Binary Number System** Base 2. Digits used: 0, 1
2	**Octal Number System** Base 8. Digits used: 0 to 7
3	**Hexadecimal Number System** Base 16. Digits used: 0 to 9, Letters used: A- F

Binary Number System

Characteristics

- Uses two digits, 0 and 1.
- Also called base 2 number system
- Each position in a binary number represents a 0 power of the base (2). Example: 2^0
- Last position in a binary number represents an x power of the base (2). Example: 2^x where x represents the last position - 1.

Example

Binary Number: 10101_2

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$
Step 3	10101_2	21_{10}

Note: 10101_2 is normally written as 10101.

Octal Number System

Characteristics

- Uses eight digits, 0,1,2,3,4,5,6,7.
- Also called base 8 number system

- Each position in an octal number represents a 0 power of the base (8). Example: 8^0
- Last position in an octal number represents an x power of the base (8). Example: 8^x where x represents the last position - 1.

Example

Octal Number – 12570_8

Calculating Decimal Equivalent –

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

Note: 12570_8 is normally written as 12570.

Hexadecimal Number System

Characteristics

- Uses 10 digits and 6 letters, 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Letters represents numbers starting from 10. A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.
- Also called base 16 number system.
- Each position in a hexadecimal number represents a 0 power of the base (16). Example 16^0.
- Last position in a hexadecimal number represents an x power of the base (16). Example 16^x where x represents the last position - 1.

Example –

Hexadecimal Number: $19FDE_{16}$

Calculating Decimal Equivalent –

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

Note – $19FDE_{16}$ is normally written as 19FDE.

There are many methods or techniques which can be used to convert numbers from one base to another. We'll demonstrate here the following –

- Decimal to Other Base System
- Other Base System to Decimal
- Other Base System to Non-Decimal
- Shortcut method – Binary to Octal
- Shortcut method – Octal to Binary
- Shortcut method – Binary to Hexadecimal
- Shortcut method – Hexadecimal to Binary

Decimal to Other Base System

Steps

- **Step 1** – Divide the decimal number to be converted by the value of the new base.
- **Step 2** – Get the remainder from Step 1 as the rightmost digit (least significant digit) of new base number.
- **Step 3** – Divide the quotient of the previous divide by the new base.
- **Step 4** – Record the remainder from Step 3 as the next digit (to the left) of the new base number.

Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.

The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.

Example −

Decimal Number: 29_{10}

Calculating Binary Equivalent −

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).

Decimal Number − 29_{10} = Binary Number − 11101_2.

Other Base System to Decimal System

Steps

- **Step 1** − Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
- **Step 2** − Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.
- **Step 3** − Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Example

Binary Number – 11101_2

Calculating Decimal Equivalent –

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number – 11101_2 = Decimal Number – 29_{10}

Other Base System to Non-Decimal System

Steps

- **Step 1** – Convert the original number to a decimal number (base 10).
- **Step 2** – Convert the decimal number so obtained to the new base number.

Example

Octal Number – 25_8

Calculating Binary Equivalent –

Step 1 – Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number – 25_8 = Decimal Number – 21_{10}

Step 2 – Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0
Step 5	1 / 2	0	1

Decimal Number − 21_{10} = Binary Number − 10101_2

Octal Number − 25_8 = Binary Number − 10101_2

Shortcut method - Binary to Octal

Steps

- **Step 1** − Divide the binary digits into groups of three (starting from the right).
- **Step 2** − Convert each group of three binary digits to one octal digit.

Example

Binary Number − 10101_2

Calculating Octal Equivalent −

Step	Binary Number	Octal Number
Step 1	10101_2	010 101
Step 2	10101_2	2_8 5_8
Step 3	10101_2	25_8

Binary Number − 10101_2 = Octal Number − 25_8

Shortcut method - Octal to Binary

Steps

- **Step 1** − Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example

Octal Number − 25_8

Calculating Binary Equivalent −

Step	Octal Number	Binary Number
Step 1	25_8	2_{10} 5_{10}
Step 2	25_8	010_2 101_2
Step 3	25_8	010101_2

Octal Number − 25_8 = Binary Number − 10101_2

Shortcut method - Binary to Hexadecimal

Steps

- **Step 1** − Divide the binary digits into groups of four (starting from the right).
- **Step 2** − Convert each group of four binary digits to one hexadecimal symbol.

Example

Binary Number − 10101_2

Calculating hexadecimal Equivalent −

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101

Step 2	10101_2	$1_{10}\ 5_{10}$
Step 3	10101_2	15_{16}

Binary Number − 10101_2 = Hexadecimal Number − 15_{16}

Shortcut method - Hexadecimal to Binary

Steps

- **Step 1** − Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example

Hexadecimal Number − 15_{16}

Calculating Binary Equivalent −

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	$1_{10}\ 5_{10}$
Step 2	15_{16}	$0001_2\ 0101_2$
Step 3	15_{16}	00010101_2

Hexadecimal Number − 15_{16} = Binary Number − 10101_2

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as **binary code**. The binary code is represented by the number as well as alphanumeric letter.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

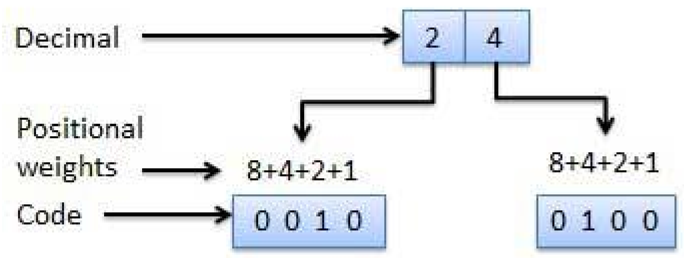
Classification of binary codes

The codes are broadly categorized into following four categories.

- Weighted Codes
- Non-Weighted Codes
- Binary Coded Decimal Code
- Alphanumeric Codes
- Error Detecting Codes
- Error Correcting Codes

Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

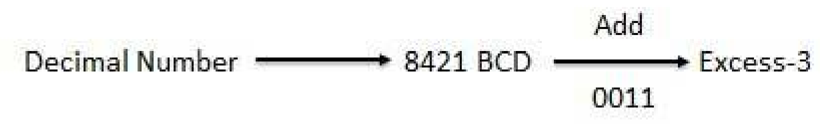


Non-Weighted Codes

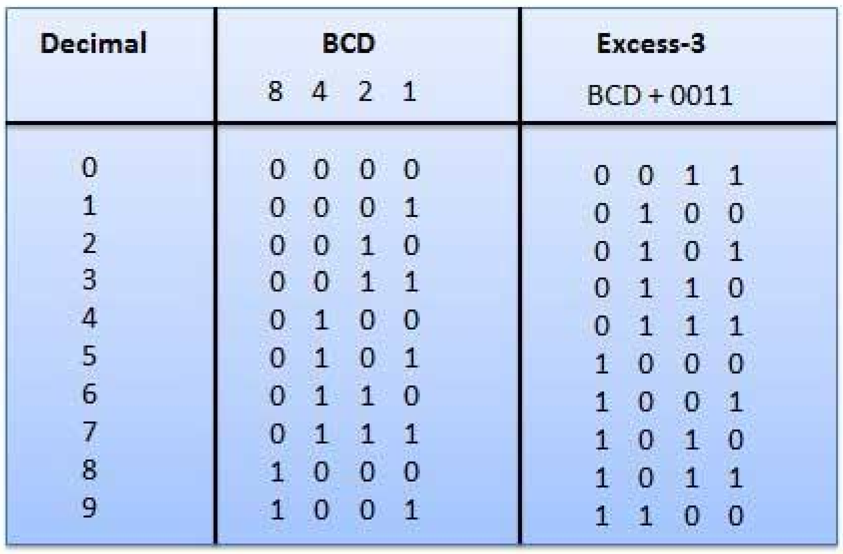
In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding (0011)2 or (3)10 to each code word in 8421. The excess-3 codes are obtained as follows –



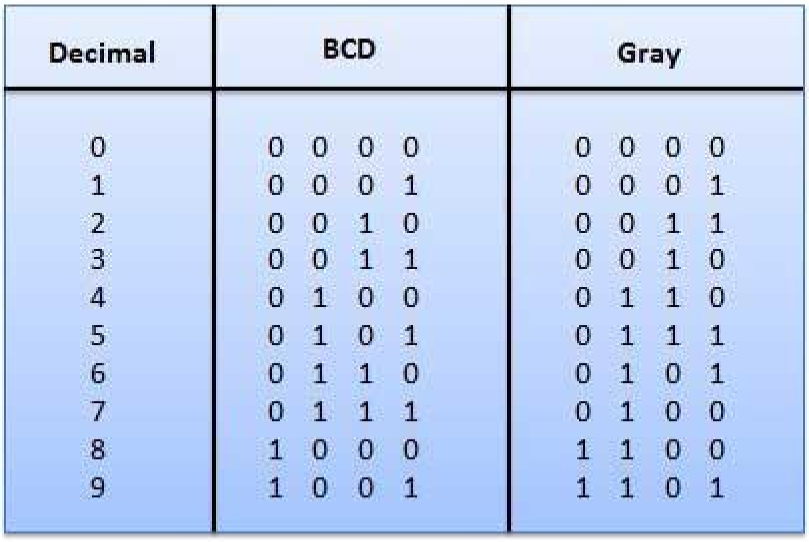
Example



Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Gray Code

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.



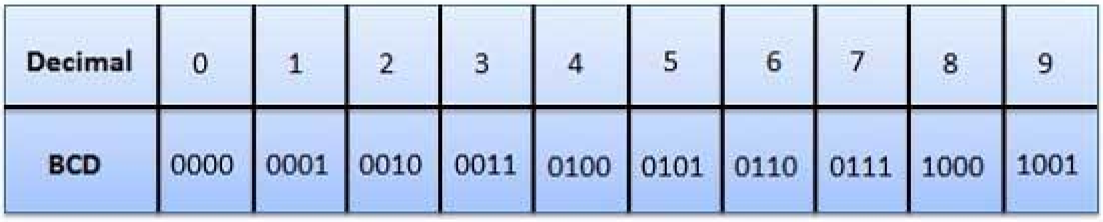
Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

Application of Gray code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

Binary Coded Decimal (BCD) code

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.



Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.

- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

Alphanumeric codes

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).
- Five bit Baudot Code.

ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

Error Codes

There are binary code techniques available to detect and correct data during data transmission.

There are many methods or techniques which can be used to convert code from one format to another. We'll demonstrate here the following

- Binary to BCD Conversion
- BCD to Binary Conversion
- BCD to Excess-3
- Excess-3 to BCD

Binary to BCD Conversion

Steps

- **Step 1** -- Convert the binary number to decimal.
- **Step 2** -- Convert decimal number to BCD.

Example − convert $(11101)_2$ to BCD.

Step 1 − Convert to Decimal

Binary Number − 11101_2

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number − 11101_2 = Decimal Number − 29_{10}

Step 2 − Convert to BCD

Decimal Number − 29_{10}

Calculating BCD Equivalent. Convert each digit into groups of four binary digits equivalent.

Step	Decimal Number	Conversion
Step 1	29_{10}	0010_2 1001_2
Step 2	29_{10}	00101001_{BCD}

Result

```
(11101)₂ =  (00101001)BCD
```

BCD to Binary Conversion

Steps

- **Step 1** -- Convert the BCD number to decimal.
- **Step 2** -- Convert decimal to binary.

Example − convert $(00101001)_{BCD}$ to Binary.

Step 1 - Convert to BCD

BCD Number − $(00101001)_{BCD}$

Calculating Decimal Equivalent. Convert each four digit into a group and get decimal equivalent for each group.

Step	BCD Number	Conversion
Step 1	$(00101001)_{BCD}$	0010_2 1001_2
Step 2	$(00101001)_{BCD}$	2_{10} 9_{10}
Step 3	$(00101001)_{BCD}$	29_{10}

BCD Number − $(00101001)_{BCD}$ = Decimal Number − 29_{10}

Step 2 - Convert to Binary

Used long division method for decimal to binary conversion.

Decimal Number − 29_{10}

Calculating Binary Equivalent −

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1

Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the least significant digit (LSD) and the last remainder becomes the most significant digit (MSD).

Decimal Number − 29_{10} = Binary Number − 11101_2

Result

```
(00101001)BCD = (11101)2
```

BCD to Excess-3

Steps

- **Step 1** -- Convert BCD to decimal.
- **Step 2** -- Add $(3)_{10}$ to this decimal number.
- **Step 3** -- Convert into binary to get excess-3 code.

Example − convert $(1001)_{BCD}$ to Excess-3.

Step 1 − Convert to decimal

$(1001)_{BCD} = 9_{10}$

Step 2 − Add 3 to decimal

$(9)_{10} + (3)_{10} = (12)_{10}$

Step 3 − Convert to Excess-3

$(12)_{10} = (1100)_2$

Result

```
(1001)BCD = (1100)XS-3
```

Excess-3 to BCD Conversion

Steps

- **Step 1** -- Subtract $(0011)_2$ from each 4 bit of excess-3 digit to obtain the corresponding BCD code.

Example – convert $(10011010)_{XS-3}$ to BCD.

```
Given XS-3 number  = 1 0 0 1 1 0 1 0
Subtract (0011)₂   = 0 0 1 1 0 0 1 1
                    --------------------
               BCD = 0 1 1 0   0 1 1 1
```

Result

```
(10011010)XS-3 = (01100111)BCD
```

Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. For each radix-r system (radix r represents base of number system) there are two types of complements.

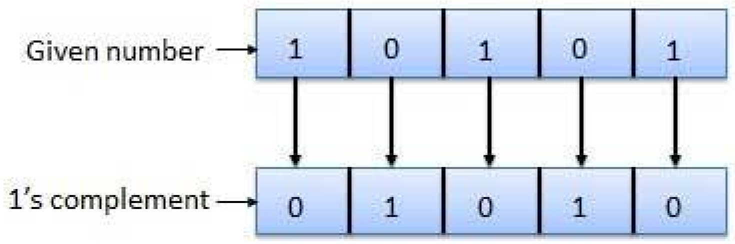
S.N.	Complement	Description
1	Radix Complement	The radix complement is referred to as the r's complement
2	Diminished Radix Complement	The diminished radix complement is referred to as the (r-1)'s complement

Binary system complements

As the binary system has base r = 2. So the two types of complements for the binary system are 2's complement and 1's complement.

1's complement

The 1's complement of a number is found by changing all 1's to 0's and all 0's to 1's. This is called as taking complement or 1's complement. Example of 1's Complement is as follows.

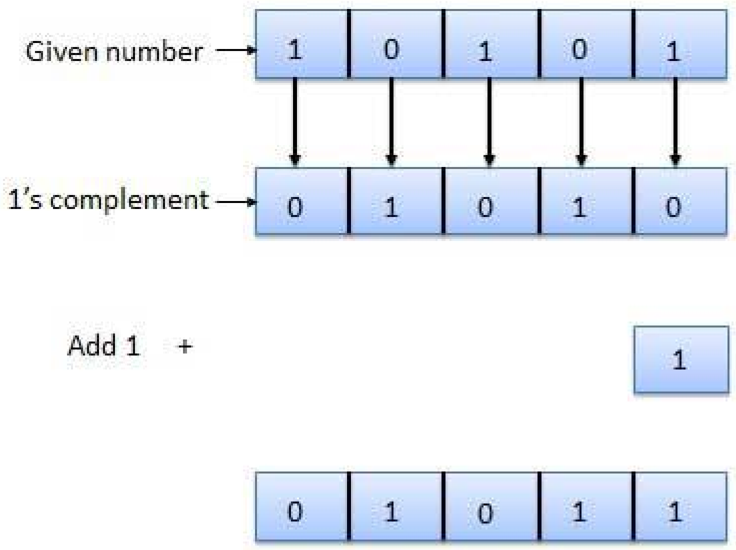


2's complement

The 2's complement of binary number is obtained by adding 1 to the Least Significant Bit (LSB) of 1's complement of the number.

2's complement = 1's complement + 1

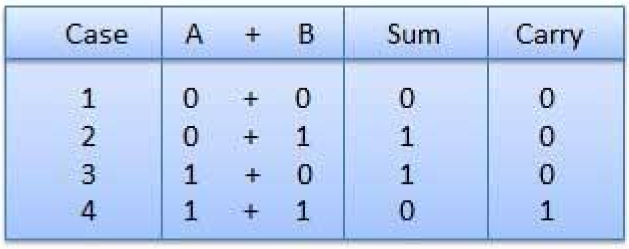
Example of 2's Complement is as follows.



Binary arithmetic is essential part of all the digital computers and many other digital system.

Binary Addition

It is a key for binary subtraction, multiplication, division. There are four rules of binary addition.



Case	A	+	B	Sum	Carry
1	0	+	0	0	0
2	0	+	1	1	0
3	1	+	0	1	0
4	1	+	1	0	1

In fourth case, a binary addition is creating a sum of (1 + 1 = 10) i.e. 0 is written in the given column and a carry of 1 over to the next column.

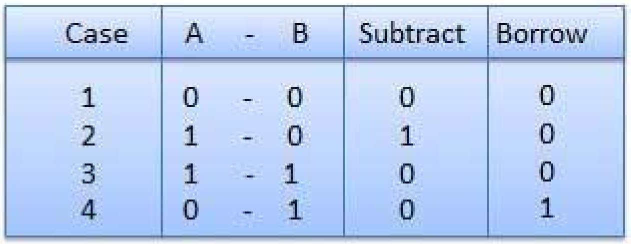
Example − Addition

0011010 + 001100 = 00100110

$$\begin{array}{rl} 1\,1 & \text{carry} \\ 0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\ +0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\ \hline 0\,1\,0\,0\,1\,1\,0 & = 38_{10} \end{array}$$

Binary Subtraction

Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There are four rules of binary subtraction.



Case	A	-	B	Subtract	Borrow
1	0	-	0	0	0
2	1	-	0	1	0
3	1	-	1	0	0
4	0	-	1	0	1

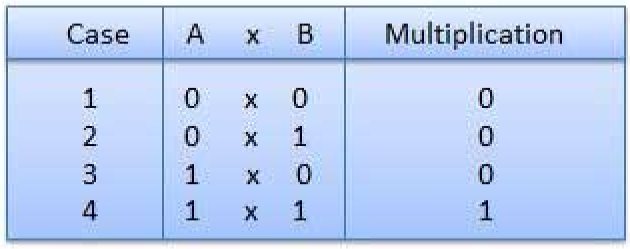
Example − Subtraction

0011010 - 001100 = 00001110

$$\begin{array}{rl} 1\,\,1 & \text{borrow} \\ 0\,0\,\cancel{1}\,\cancel{1}\,0\,1\,0 & = 26_{10} \\ -0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\ \hline 0\,0\,0\,1\,1\,1\,0 & = 14_{10} \end{array}$$

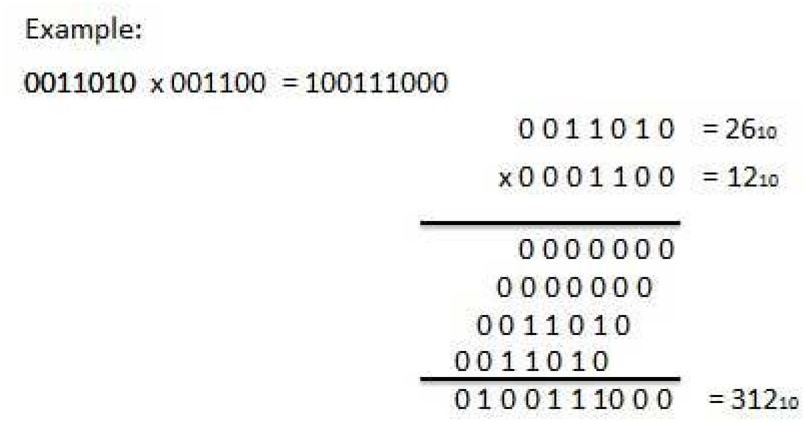
Binary Multiplication

Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved. There are four rules of binary multiplication.



Case	A	x	B	Multiplication
1	0	x	0	0
2	0	x	1	0
3	1	x	0	0
4	1	x	1	1

Example − Multiplication



Example:

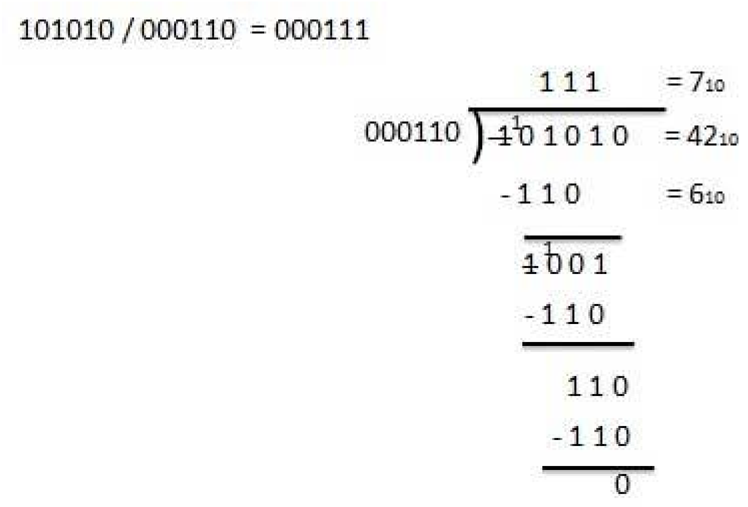
0011010 x 001100 = 100111000

```
        0 0 1 1 0 1 0    = 26₁₀
      x 0 0 0 1 1 0 0    = 12₁₀
   ____________________
        0 0 0 0 0 0 0
      0 0 0 0 0 0 0
    0 0 1 1 0 1 0
  0 0 1 1 0 1 0
   ____________________
  0 1 0 0 1 1 1 0 0 0    = 312₁₀
```

Binary Division

Binary division is similar to decimal division. It is called as the long division procedure.

Example − Division



101010 / 000110 = 000111

```
           1 1 1       = 7₁₀
        ___________
000110 ) 1 0 1 0 1 0   = 42₁₀
        -1 1 0         = 6₁₀
         _____
         1 0 0 1
        -1 1 0
         _______
           1 1 0
          -1 1 0
          ______
               0
```

Hexadecimal Number System

Following are the characteristics of a hexadecimal number system.

- Uses 10 digits and 6 letters, 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Letters represents numbers starting from 10. A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.
- Also called base 16 number system.
- Each position in a hexadecimal number represents a 0 power of the base (16). Example − 16^0
- Last position in a hexadecimal number represents an x power of the base (16). Example − 16^x where x represents the last position - 1.

Example

Hexadecimal Number − $19FDE_{16}$

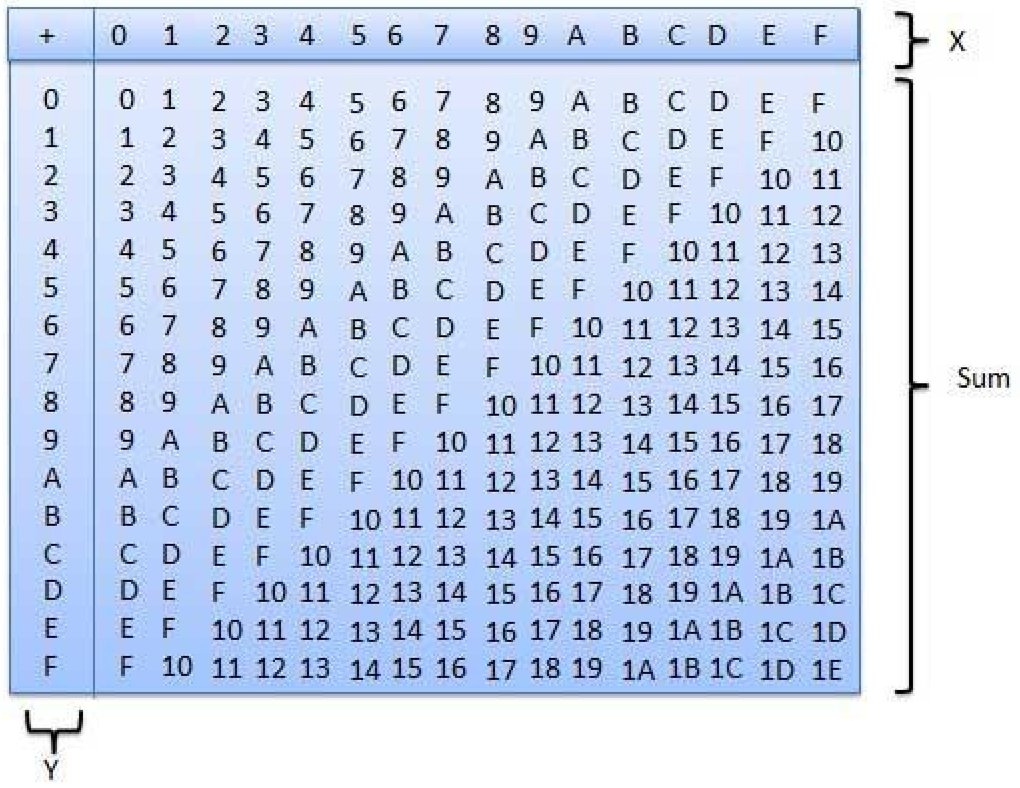
Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

Note – $19FDE_{16}$ is normally written as 19FDE.

Hexadecimal Addition

Following hexadecimal addition table will help you greatly to handle Hexadecimal addition.



+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

To use this table, simply follow the directions used in this example − Add A_{16} and 5_{16}. Locate A in the X column then locate the 5 in the Y column. The point in 'sum' area where these two columns intersect is the sum of two numbers.

```
A16 + 516 = F16.
```

Example − Addition

$4A6_{16} + 1B3_{16} = 659_{16}$

$$\begin{array}{rl} 1 & \text{carry} \\ 4\,A\,6 & = 1190_{10} \\ +\,1\,B\,3 & = 435_{10} \\ \hline 6\,5\,9 & = 1625_{10} \end{array}$$

Hexadecimal Subtraction

The subtraction of hexadecimal numbers follow the same rules as the subtraction of numbers in any other number system. The only variation is in borrowed number. In the decimal system, you borrow a group of 10_{10}. In the binary system, you borrow a group of 2_{10}. In the hexadecimal system you borrow a group of 16_{10}.

Example - Subtraction

$4A6_{16} - 1B3_{16} = 2F3_{16}$

$$\begin{array}{rl}
16 & \text{borrow} \\
{}^{3}4\,A\,6 & = 1190_{10} \\
-\,1\,B\,3 & = 435_{10} \\
\hline
2\,F\,3 & = 755_{10}
\end{array}$$

Boolean Algebra is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as **Binary Algebra** or **logical Algebra**. Boolean algebra was invented by **George Boole** in 1854.

Rule in Boolean Algebra

Following are the important rules used in Boolean algebra.

- Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
- Complement of a variable is represented by an overbar (-). Thus, complement of variable B is represented as $\overline{B}$. Thus if B = 0 then $\overline{B}$ = 1 and B = 1 then $\overline{B}$ = 0.
- ORing of the variables is represented by a plus (+) sign between them. For example ORing of A, B, C is represented as A + B + C.
- Logical ANDing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometime the dot may be omitted like ABC.

Boolean Laws

There are six types of Boolean Laws.

Commutative law

Any binary operation which satisfies the following expression is referred to as commutative operation.

(i) $A.B = B.A$ (ii) $A + B = B + A$

Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

Associative law

This law states that the order in which the logic operations are performed is irrelevant as their effect is the same.

(i) $(A.B).C = A.(B.C)$ (ii) $(A + B) + C = A + (B + C)$

Distributive law

Distributive law states the following condition.

$A.(B + C) = A.B + A.C$

AND law

These laws use the AND operation. Therefore they are called as **AND** laws.

(i) $A.0 = 0$ (ii) $A.1 = A$

(iii) $A.A = A$ (iv) $A.\overline{A} = 0$

OR law

These laws use the OR operation. Therefore they are called as **OR** laws.

(i) $A + 0 = A$ (ii) $A + 1 = 1$

(iii) $A + A = A$ (iv) $A + \overline{A} = 1$

INVERSION law

This law uses the NOT operation. The inversion law states that double inversion of a variable results in the original variable itself.

$\overline{\overline{A}} = A$

LOGIC GATES

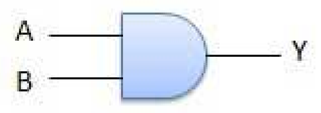
Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a **certain logic**. Based on this, logic gates are named as AND gate, OR gate, NOT gate etc.

AND Gate

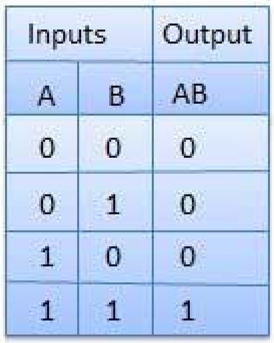
A circuit which performs an AND operation is shown in figure. It has n input (n >= 2) and one output.

Y = A AND B AND C .... N
Y = A.B.C .... N
Y = ABC .... N

Logic diagram



Truth Table



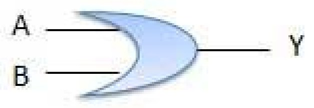
Inputs		Output
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

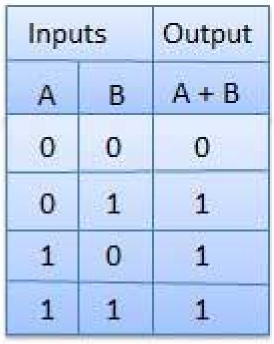
A circuit which performs an OR operation is shown in figure. It has n input (n >= 2) and one output.

Y = A OR B OR C .... N
Y = A + B + C .... N

Logic diagram



Truth Table



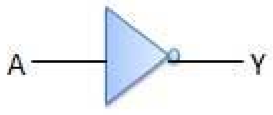
Inputs		Output
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

NOT gate is also known as **Inverter**. It has one input A and one output Y.

Y	=	NOT A
Y	=	$\overline{A}$

Logic diagram



Truth Table

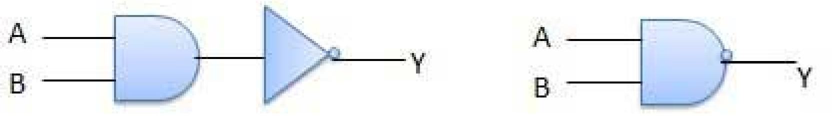
Inputs	Output
A	B
0	1
1	0

NAND Gate

A NOT-AND operation is known as NAND operation. It has n input (n >= 2) and one output.

Y	=	A NOT AND B NOT AND C .... N
Y	=	A NAND B NAND C .... N

Logic diagram



Truth Table

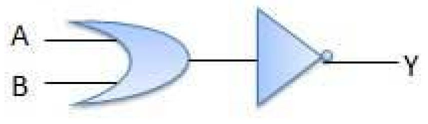
Inputs		Output
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

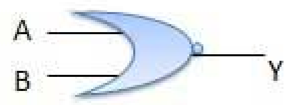
NOR Gate

A NOT-OR operation is known as NOR operation. It has n input (n >= 2) and one output.

Y	=	A NOT OR B NOT OR C .... N
Y	=	A NOR B NOR C .... N

Logic diagram





Truth Table

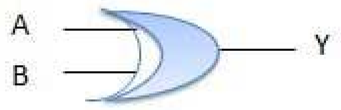
Inputs		Output
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate

XOR or Ex-OR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-OR gate is abbreviated as EX-OR gate or sometime as X-OR gate. It has n input (n >= 2) and one output.

Y	=	A XOR B XOR C .... N
Y	=	$A \oplus B \oplus C$ .... N
Y	=	$A\overline{B} + \overline{A}B$

Logic diagram



Truth Table

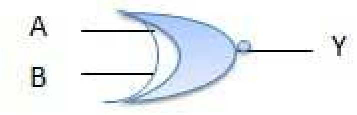
Inputs		Output
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

XNOR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-NOR gate is abbreviated as EX-NOR gate or sometime as X-NOR gate. It has n input (n >= 2) and one output.

$$Y = A \text{ XOR } B \text{ XOR } C \ldots\ldots N$$
$$Y = A \odot B \odot C \ldots\ldots N$$
$$Y = \overline{A}\,\overline{B} + AB$$

Logic diagram



Truth Table

Inputs		Output
A	B	$A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

UNIT II
Gate Level Minimization:

Introduction to Karnaugh Maps

The Karnaugh map (or K-map) is a visual way of detecting redundancy in the SoP.

The K-map can be easily used for circuits with 2, 3, or 4 inputs.

It consists of an array of cells, each representing a possible combination of inputs.

- The cells are arranged to that each cell's input combination differs from adjacent cells by only a single bit.
- This is called Gray code ordering – it ensures that physical neighbours is the array are logical neighbou rs as well. (In other words, neighbouring bit patterns are nearly the same, differing by only 1 bit).

Consider the following arrangements of cells:

2-input

a'@ b' 00	a'@ b 01
a @ b' 10	a @ b 11

3-input

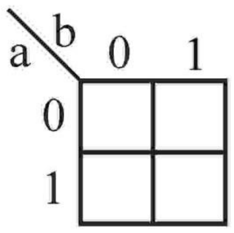
a'@ b'@ c' 000	a'@ b'@ c 001	a'@ b @ c 011	a'@ b @ c' 010
a @ b'@ c' 100	a @ b'@ c 101	a @ b @ c 111	a @ b @ c' 110

4-input

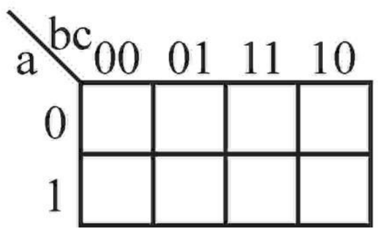
a'@b'@c'@d' 0000	a'@b'@c'@d 0001	a'@b'@c@d 0011	a'@b'@c@d' 0010
a'@b@c'@d' 0100	a'@b@c'@d 0101	a'@b@c@d 0111	a'@b@c@d' 0110
a@b@c'@d' 1100	a@b@c'@d 1101	a@b@c@d 1111	a@b@c@d' 1110
a@b'@c'@d' 1000	a@b'@c'@d 1001	a@b'@c@d 1011	a@b'@c@d' 1010

The cells are arranged as above, but we write them empty, like this:

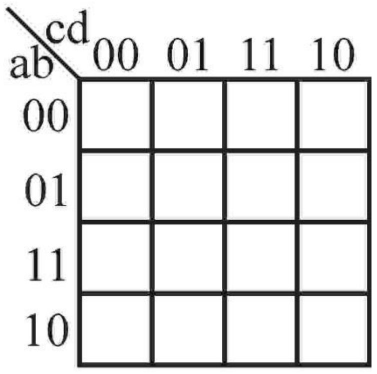
2-input:



3-input:



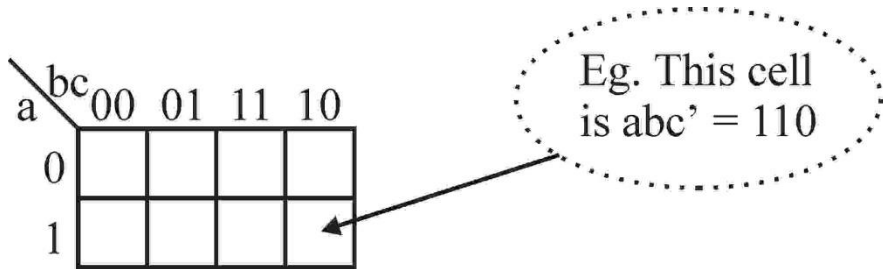
4-input:



Note that the numbers are *not* in binary order, but are arranged so that only a single bit changes between neighbours.

This one-bit change applies at the edges, too. So cells in the same row on the left and right edges of the array also only differ by one bit.

Note: The value of a particular cell is found by combining the numbers at the edges of the row and column.



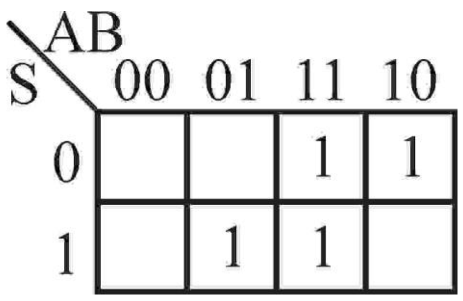
Also, in general, it is easier to order the inputs to a K-map so that they can be read like a binary number. *(Show example.)*

So, we have this grid. What do w e do with it?

- We put 1's in all the cells that represent minterms in the SSoP . (In other w ords, we find the 1's in the truth table output, and put 1's in the cells corresponding to the same inputs.)

Let's do this in relation to the 2-input multiplexer example:

S	A	B	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

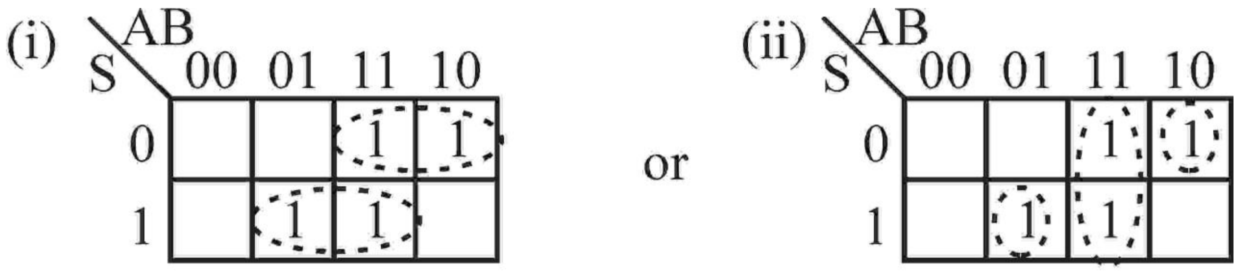


If there are tw o neigh bouring 1's in the grid, it means that the input bit change between the tw o cells has no effect on the output, and thu s there is redundancy. This leads to a basic strategy.

Basic Strategy:
Group adjacent 1's together in square or rectangular groups of 2, 4, 8, or 16, such that the total number of groups and isolated 1's is minimized, while using as large groups as possible. Groups may overlap, so that a particular cell may be included in more than one group.

(Recall that adjacency wrap s around edges of grid.)

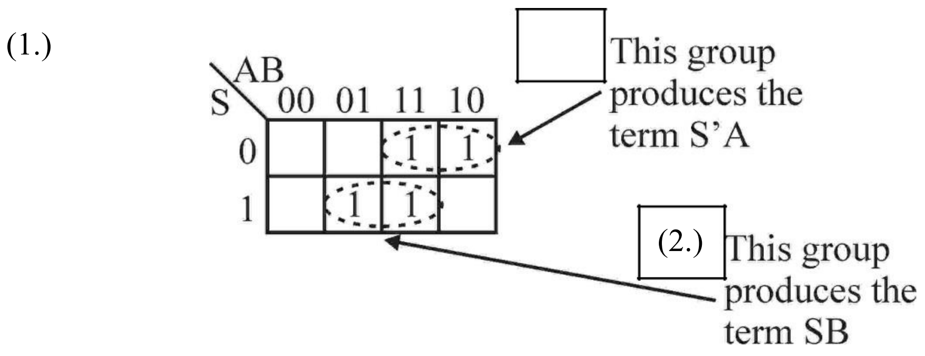
Applying this to the multiplexer example:



Option (i) is best since only 2 groups vs. 3

So, considering the best option above (i), notice the following:

1. B changes but the output doesn't, so B is redundant in this group (See comment 1, below).
2. A changes but the output doesn't, so A is redundant in this group (See comment 2, below).


(1.)
AB
S
00 01 11 10
0
1
1 1
1 1
This group produces the term S'A
(2.)
This group produces the term SB

So, we write out Boolean expressions for each group, leaving out the redundant elements. That is, for each group, we write out the inputs that don't change.

The multiplexer example, with two groups, gives us two terms, Y
= S@B + S'@A
which is the same as what we achieved through using Boolean algebra to reduce the circuit.

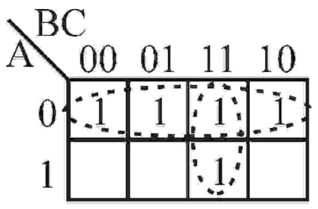
So, we can summarize this process into a basic set of rules:

Rules for K-Maps

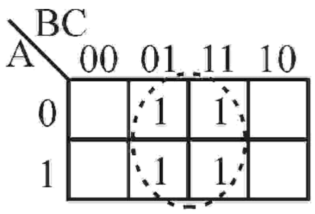
1. Each cell with a 1 must be included in at least one group.
2. Try to form the largest possible groups.
3. Try to end up with as few groups as possible.
4. Groups may be in sizes that are powers of 2: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, ...
5. Groups may be square or rectangular only (including wrap-around at the grid edges). No diagonals or zig-zags can be used to form a group.
6. The larger a group is, the more redundant inputs there are:
   i. A group of 1 has no redundant inputs.
   ii. A group of 2 has 1 redundant input.
   iii. A group of 4 has 2 redundant inputs.
   iv. A group of 8 has 3 redundant inputs.
   v. A group of 16 has 4 redundant inputs.

The following simple examples illustrate rule 6 above.

Eg.

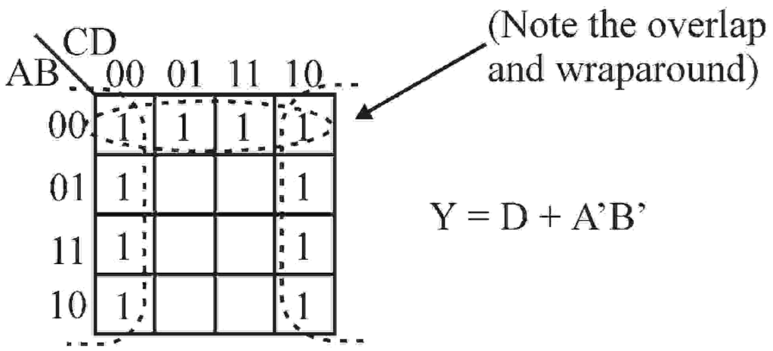


Y = A' + BC



Y = C

Groups of 8 are similar....

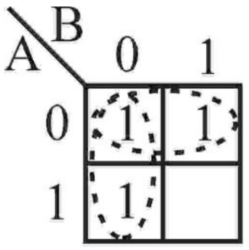


Y = D + A'B'

Examples

2-input Example

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



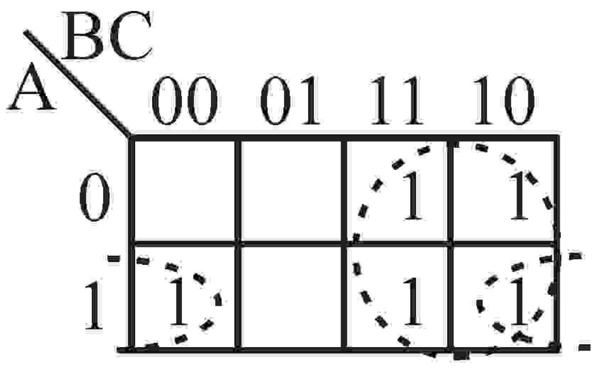
Y = B' + A'

(This is NAND)

Direct from truth table: Y = A'B' + A'B + AB'

3-input Example

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

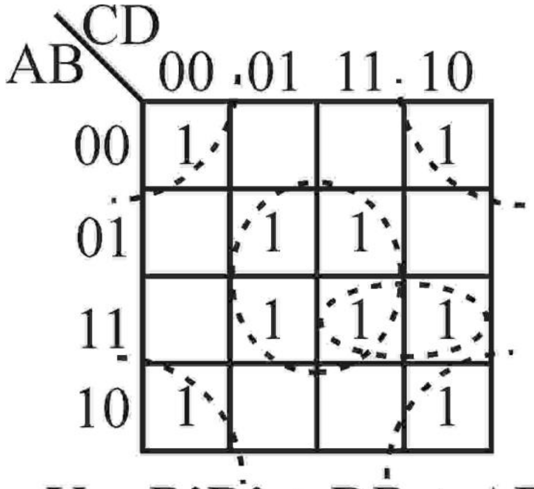


Y = AC’ + B

Direct from truth table: Y = A’BC’ + A’BC + AB’C’ + ABC’ + ABC

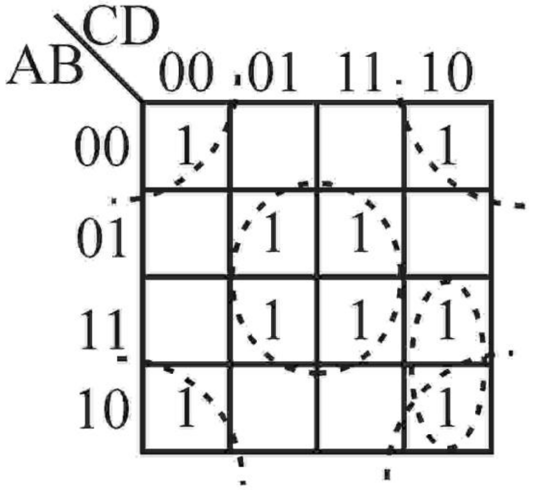
4-input Example

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



Y = B'D' + DB + ABC

or

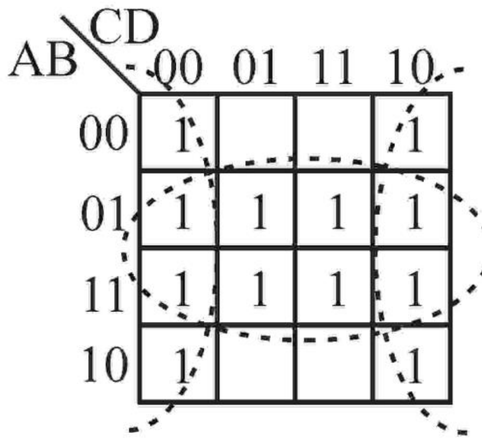


Y = B'D' + DB + ACD'

use a K-map to reduce the following 4-input circuit.

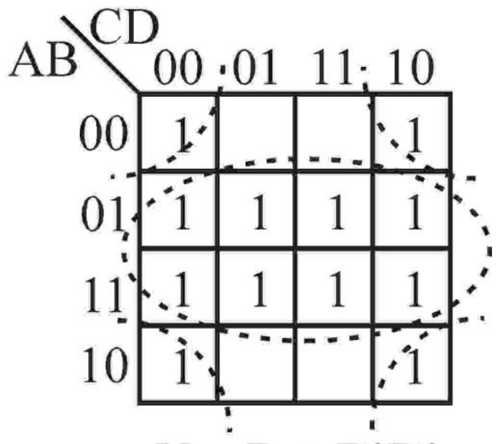
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Best solution:



Y = D' + B

Alternate solution:



Y = B + B'D'

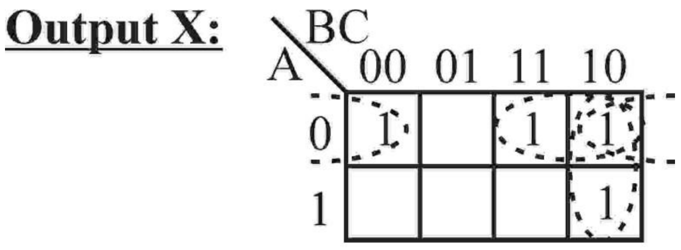
Using K-Maps

So, we have seen how K-maps can help us minimize logic.

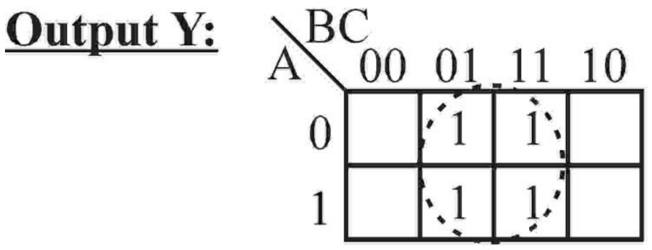
How do we apply this to a circuit which has multiple outputs?

- We simply use multiple K -maps, and treat them as separate circuits that just happen to have the same inputs.

A	B	C	X	Y
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1



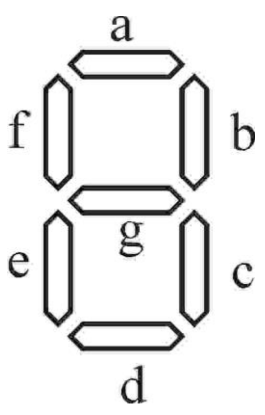
X = A'C' + A'B + BC'



Y = C

Now, what if there are cases w here our output doesn't matter?
• In other words, what if there are input combinations for which we don't care what the output is?

You might wonder when such a case might occur. So, consider a seven-segment display (like in many alarm clocks, or the classroom wall clock).



On this seven-segment display, we only wish to display the numbers 0 to 9.

- To represent the numbers 0 to 9, we need 4 bits: 0000 represents zero, 1001 represents nine, and the other digits are similarly represented.
- But 4 bits can actually represent values from 0 to 15!
- So, when input bit values are from 10 to 15, we don't care what the output is. (That is, we would never expect the input to the circuit to have those values, so we don't care what the circuit does if it does happen).

This concept of not caring allows us to minimize the logic further.

So, a circuit to control a seven-segment display would have 4 inputs (the binary number representation of the decimal number to be displayed) and 7 outputs (control signals to each of the 7 segments that tell them to light up or turn off).

Consider the topmost segment, labeled 'a'.

- assume that a value of 1 activates the segment, and a value of 0 turns it off / leaves it unlit
- the top segment will be lit for values 0 , 2, 3, 5, 6, 7, 8, 9
- we don't care what it does if the values input happen to be in the range of 10 to 15

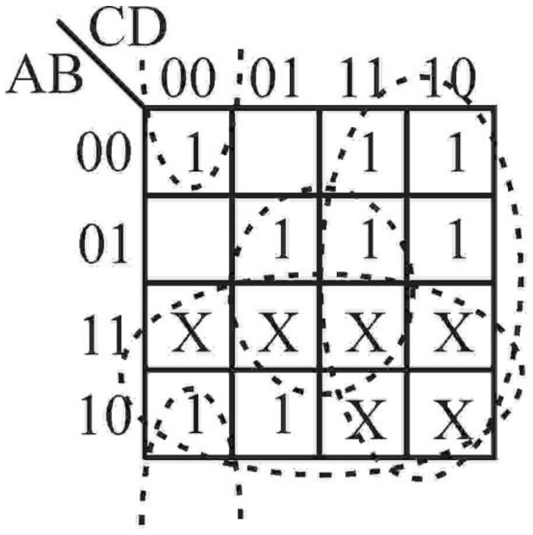
In both the truth tab le and K-map, we represent a "don't care" with an X . We draw them on the K-map grid along with the 1's.

Then, when forming groups on the K-map, we can treat the "don't cares" (the X cells) as 1's. This may often allow us to make larger groups, and thus reduce the logic more.

So, we can include "don't cares" in groups, but we never have groups of *just* don't cares. Basically, we can treat them as 1's or 0's, whichever is most convenient.

Truth table for output a of seven-segment display.

A	**B**	**C**	**D**	**a**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



a = A + C + BD + B'C'D'

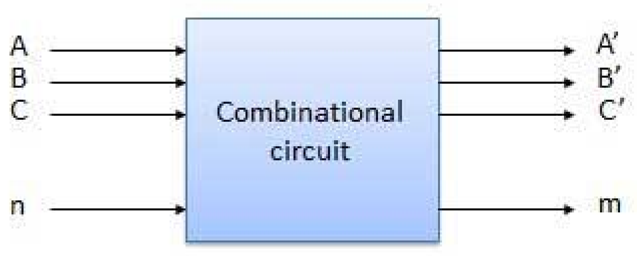
Thus, using "don't care" outputs can sometimes allow us to further minimize logic, but only in cases where we really don't care. You should look to utilize "don't cares" if the design specification allows it, but unfortunately that won't always be the case.

UNIT III
COMBINATIONAL LOGIC CIRCUITS:

Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −

- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

Block diagram

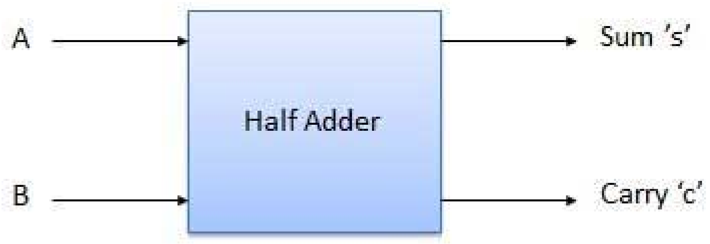


We're going to elaborate few important combinational circuits as follows.

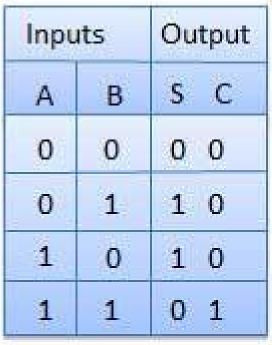
Half Adder

Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two **single** bit numbers. This circuit has two outputs **carry** and **sum**.

Block diagram

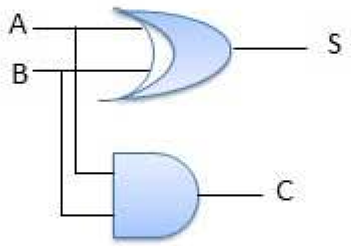


Truth Table



Inputs		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

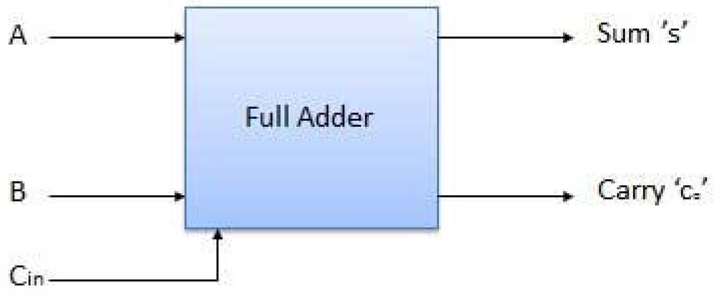
Circuit Diagram



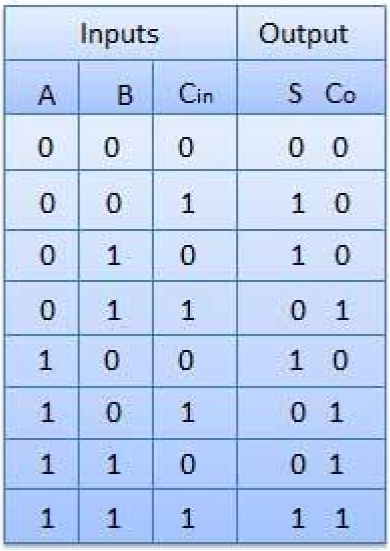
Full Adder

Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.

Block diagram

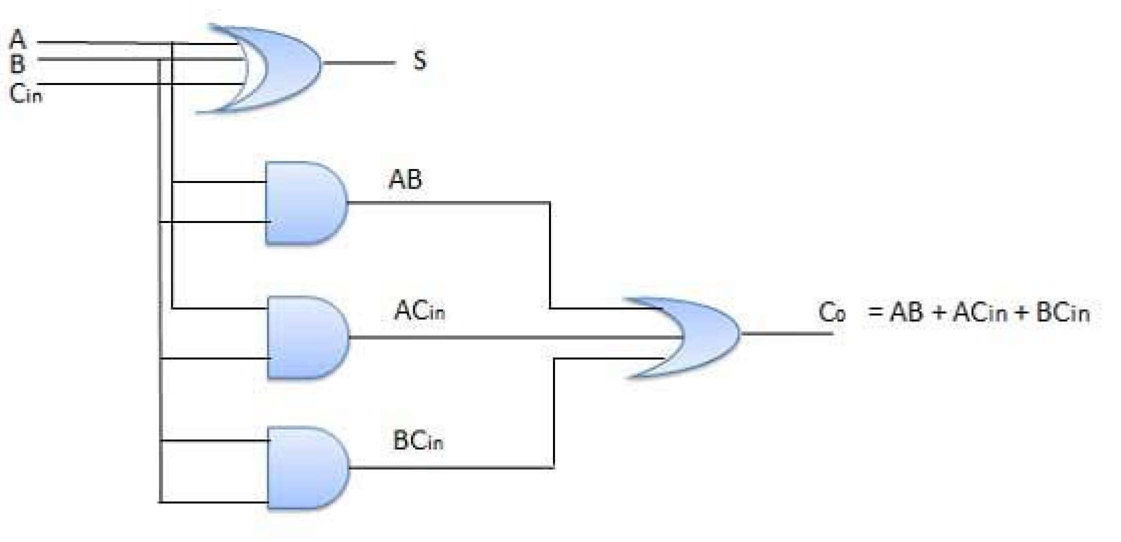


Truth Table



Inputs			Output	
A	B	Cin	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Circuit Diagram



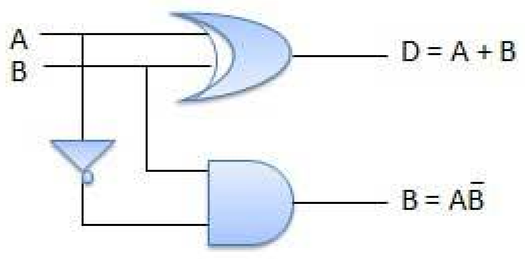
Half-Subtractors

Half-subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.

Truth Table

Inputs		Output	
A	B	(A – B)	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

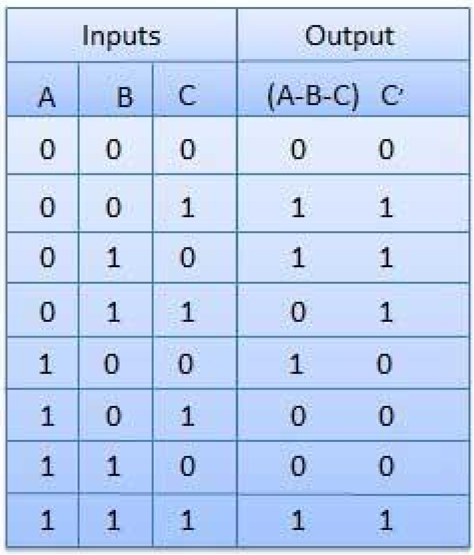
Circuit Diagram



Full-Subtractors

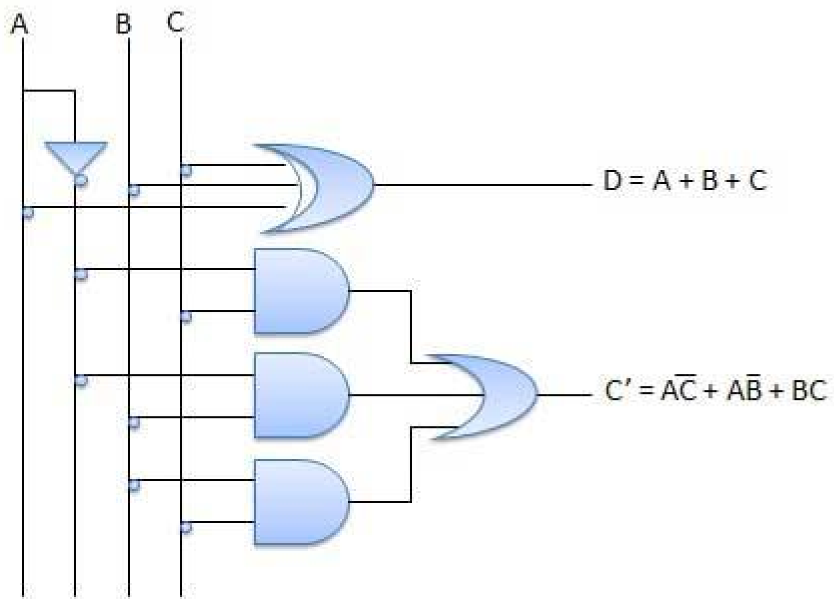
The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A,B,C and two output D and C'. A is the 'minuend', B is 'subtrahend', C is the 'borrow' produced by the previous stage, D is the difference output and C' is the borrow output.

Truth Table



Inputs			Output	
A	B	C	(A-B-C)	C'
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Circuit Diagram



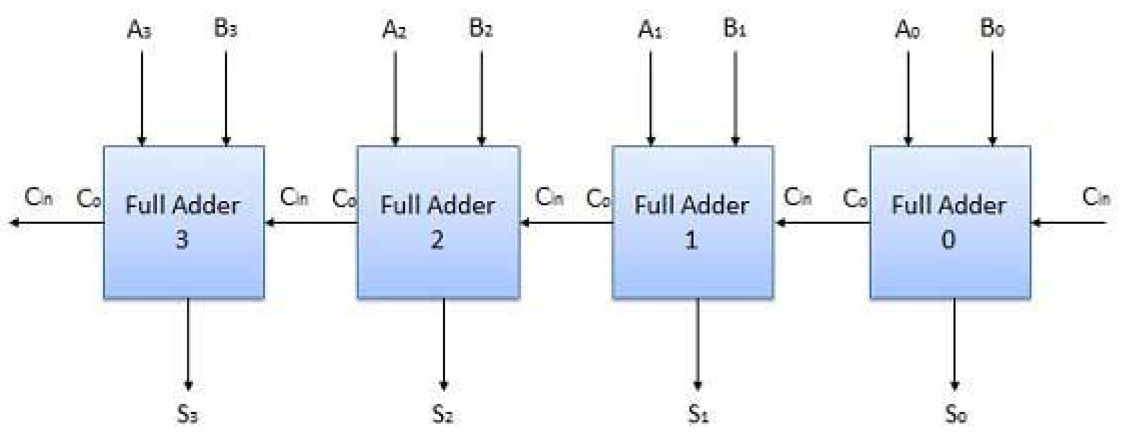
N-Bit Parallel Adder

The Full Adder is capable of adding only two single digit binary number along with a carry input. But in practical we need to add binary numbers which are much longer than just one bit. To add two n-bit binary numbers we need to use the n-bit parallel adder. It uses a number of full adders in cascade. The carry output of the previous full adder is connected to carry input of the next full adder.

4-Bit Parallel Adder

In the block diagram, A_0 and B_0 represent the LSB of the four bit words A and B. Hence Full Adder-0 is the lowest stage. Hence its C_{in} has been permanently made 0. The rest of the connections are exactly same as those of n-bit parallel adder is shown in fig. The four bit parallel adder is a very common logic circuit.

Block diagram



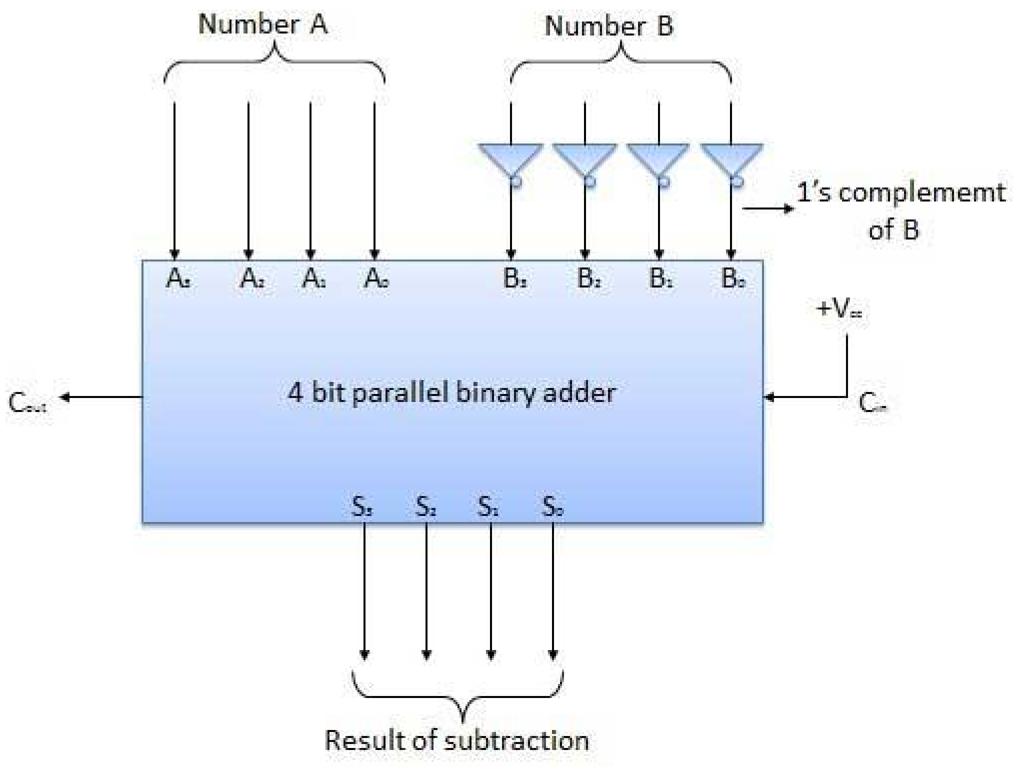
N-Bit Parallel Subtractor

The subtraction can be carried out by taking the 1's or 2's complement of the number to be subtracted. For example we can perform the subtraction (A-B) by adding either 1's or 2's complement of B to A. That means we can use a binary adder to perform the binary subtraction.

4 Bit Parallel Subtractor

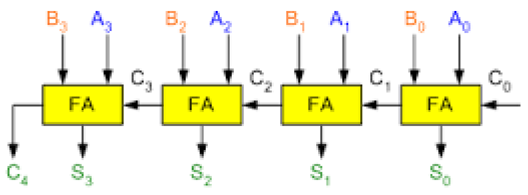
The number to be subtracted (B) is first passed through inverters to obtain its 1's complement. The 4-bit adder then adds A and 2's complement of B to produce the subtraction. S_3 S_2 S_1 S_0 represents the result of binary subtraction (A-B) and carry output C_{out} represents the polarity of the result. If A > B then Cout = 0 and the result of binary form (A-B) then C_{out} = 1 and the result is in the 2's complement form.

Block diagram



Carry Look Ahead Adder

In ripple carry adders, the carry propagation time is the major speed limiting factor.



B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0
C_3 C_2 C_1 C_0
FA FA FA FA
C_4 S_3 S_2 S_1 S_0

Most other arithmetic operations, e.g. multiplication and division are implemented using several add/subtract steps. Thus, improving the speed of addition will improve the speed of all other arithmetic operations.

Accordingly, reducing the carry propagation delay of adders is of great importance. Different logic design approaches have been employed to overcome the carry propagation problem.

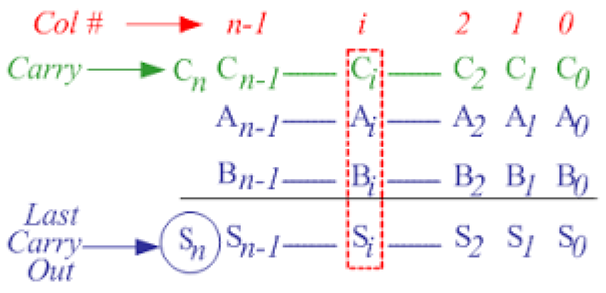
One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.

This type of adder circuit is called as carry look-ahead adder (CLA adder). It is based on the fact that a carry signal will be generated in two cases:

(1) when both bits A_i and B_i are 1, or

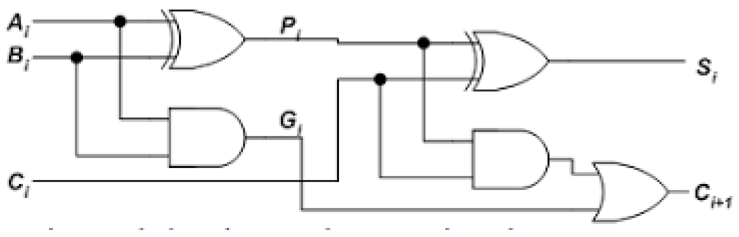
(2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

To understand the carry propagation problem, let’s consider the case of adding two n-bit numbers A and B.



Col #
$n-1$ i 2 1 0
Carry
C_n C_{n-1} C_i C_2 C_1 C_0
A_{n-1} A_i A_2 A_1 A_0
B_{n-1} B_i B_2 B_1 B_0
Last Carry Out
S_n S_{n-1} S_i S_2 S_1 S_0

The Figure shows the full adder circuit used to add the operand bits in the ith column; namely A_i & B_i and the carry bit coming from the previous column (C_i).



In this circuit, the 2 internal signals P_i and G_i are given by:

$$P_i = A_i \oplus B_i \text{ ....}(1)$$

$$G_i = A_i B_i \text{ ....}(2)$$

The output sum and carry can be defined as :

$$S_i = P_i \oplus C_i \text{ ....}(3)$$

$$C_{i+1} = G_i + P_i C_i \text{ ....}(4)$$

G_i is known as the carry Generate signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i).

P_i is known as the carry propagate signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0).

Computing the values of P_i and G_i only depend on the input operand bits (A_i & B_i) as clear from the Figure and equations.

Thus, these signals settle to their steady-state value after the propagation through their respective gates.

Computed values of all the P_i's are valid one XOR-gate delay after the operands A and B are made valid.

Computed values of all the G_i's are valid one AND-gate delay after the operands A and B are made valid.

The Boolean expression of the carry outputs of various stages can be written as follows:

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$

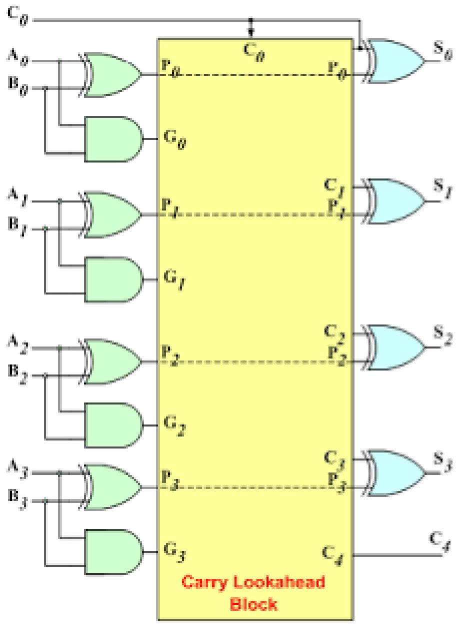
$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3$

$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

In general, the ith carry output is expressed in the form $\mathbf{C_i} = F_i$ (P's, G's , $\mathbf{C_0}$).
In other words, each carry signal is expressed as a direct SOP function of $\mathbf{C_0}$ rather than its preceding carry signal.
Since the Boolean expression for each output carry is expressed in SOP form, it can be implemented in two-level circuits.
The 2-level implementation of the carry signals has a propagation delay of 2 gates, i.e., 2τ.
The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:



First level: Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate). Output signals of this level (P's & G's) will be valid after 1τ.
Second level: The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals (C_1, C_2, C_3, and C_4) as defined by the above expressions. Output signals of this level (C_1, C_2, C_3, and C_4) will be valid after 3τ.
Third level: Four XOR gates which generate the sum signals (S_i) ($S_i = P_i \oplus C_i$). Output signals of this level (S_0, S_1, S_2, and S_3) will be valid after 4τ.

Thus, the 4 Sum signals (S_0, S_1, S_2 & S_3) will all be valid after a total delay of 4τ compared to a delay of (2n+1) τ for Ripple Carry adders.
For a 4-bit adder (n = 4), the Ripple Carry adder delay is 9τ.

The disadvantage of the CLA adders is that the carry expressions (and hence logic) become quite complex for more than 4 bits.

Thus, CLA adders are usually implemented as 4-bit modules that are used to build larger size adders.

BCD Adder

If two BCD digits are added then their sum result will not always be in BCD.
Consider the two given examples.

Correct: Result is in BCD.

$$\begin{array}{r@{\;=\;}r} 0110 & 6 \\ +\underline{0011} & +\underline{3} \\ \underline{1001} & \underline{9} \end{array}$$

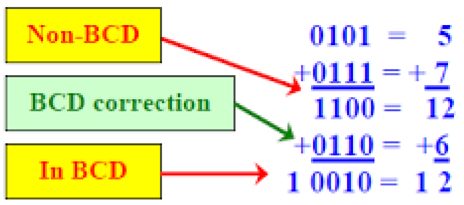
Wrong: Result is not in BCD.

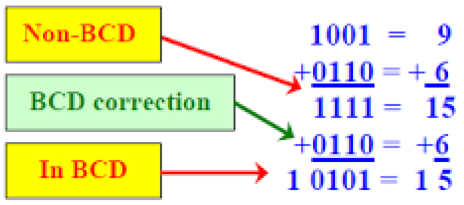
$$\begin{array}{r@{\;=\;}r} 0101 & 5 \\ +\underline{0111} & +\underline{7} \\ \underline{1100} & \underline{12} \end{array}$$

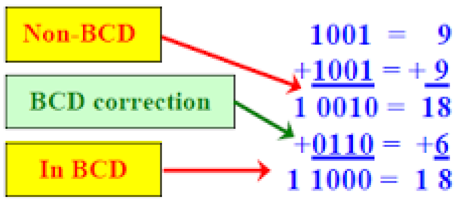
In the first example, result is in BCD while in the second example it is not in BCD.
Four bits are needed to represent all BCD digits (0 – 9). But with four bits we can represent up to 16 values (0000 through 1111). The extra six values (1010 through 1111) are not valid BCD digits.
Whenever the sum result is > 9, it will not be in BCD and will require correction to get a valid BCD result.

Z_3	Z_2	Z_1	Z_0		F
0	0	0	0		0
0	0	0	1		0
0	0	1	0		0
0	0	1	1		0
0	1	0	0		0
0	1	0	1		0
0	1	1	0		0
0	1	1	1		0
1	0	0	0		0
1	0	0	1		0
1	0	1	0		1
1	0	1	1		1
1	1	0	0		1
1	1	0	1		1
1	1	1	0		1
1	1	1	1		1

Correction is done through the addition of 6 to the result to skip the six invalid values as shown in the truth table by yellow color.
Consider the given examples of non-BCD sum result and its correction.







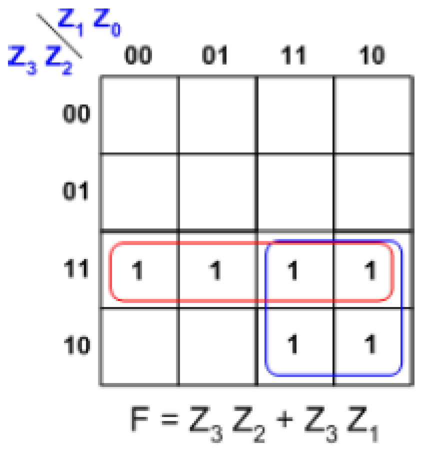
A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum BCD digit and a carry out bit.

The maximum sum result of a BCD input adder can be 19. As maximum number in BCD is 9 and may be there will be a carry from previous stage also, so 9 + 9 + 1 = 19

The following truth table shows all the possible sum results when two BCD digits are added.

Dec	CO	Z_3	Z_2	Z_1	Z_0		F
0	0	0	0	0	0		0
1	0	0	0	0	1		0
2	0	0	0	1	0		0
3	0	0	0	1	1		0
4	0	0	1	0	0		0
5	0	0	1	0	1		0
6	0	0	1	1	0		0
7	0	0	1	1	1		0
8	0	1	0	0	0		0
9	0	1	0	0	1		0
10	0	1	0	1	0		1
11	0	1	0	1	1		1
12	0	1	1	0	0		1
13	0	1	1	0	1		1
14	0	1	1	1	0		1
15	0	1	1	1	1		1
16	1	0	0	0	0		1
17	1	0	0	0	1		1
18	1	0	0	1	0		1
19	1	0	0	1	1		1

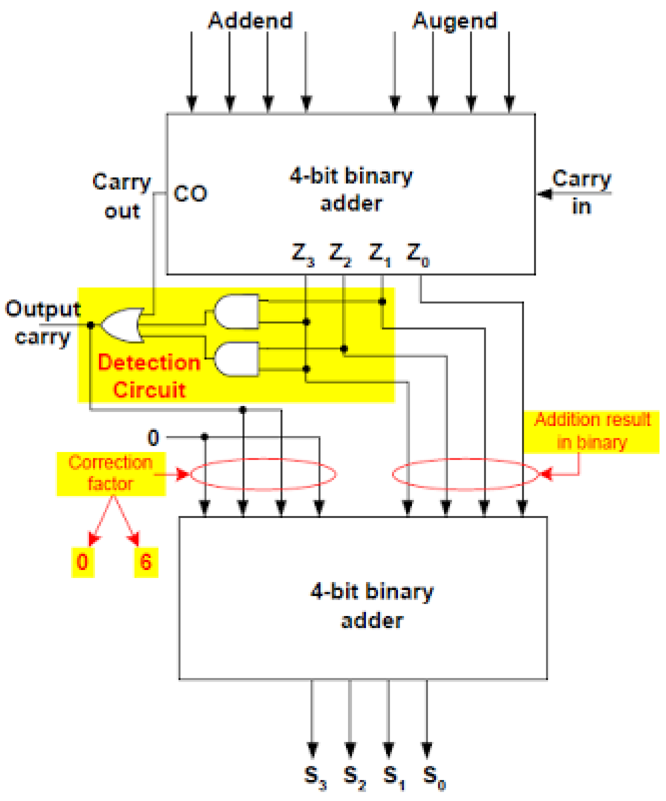
The logic circuit that checks the necessary BCD correction can be derived by detecting the condition where the resulting binary sum is 01010 through 10011 (decimal 10 through 19).
It can be done by considering the shown truth table, in which the function F is true when the digit is not a valid BCD digit. It can be simplified using a 5-variable K-map. But detecting values 1010 through 1111 (decimal 10 through 15) can also be done by using a 4-variable K-map as shown in the figure.



Values greater than 1111, i.e., from 10000 through 10011 (decimal 16 through 19) can be detected by the carry out (CO) which equals 1 only for these output values. So, F = CO = 1 for these values. Hence, F is true when CO is true OR when ($Z_3 Z_2 + Z_3 Z_1$) is true.
Thus, the correction step (adding 0110) is performed if the following function equals 1:

$$F = CO + Z_3 Z_2 + Z_3 Z_1$$

The circuit of the BCD adder will be as shown in the figure.



The two BCD digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. The bottom 4-bit binary adder is used to add the correction factor to the binary result of the top binary adder.

Note:

1. When the Output carry is equal to zero, the correction factor equals zero.
2. When the Output carry is equal to one, the correction factor is 0110.

The output carry generated from the bottom binary adder is ignored, since it supplies information already available at the output-carry terminal.

A decimal parallel adder that adds n decimal digits needs n BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher-order stage.

Binary Multiplier

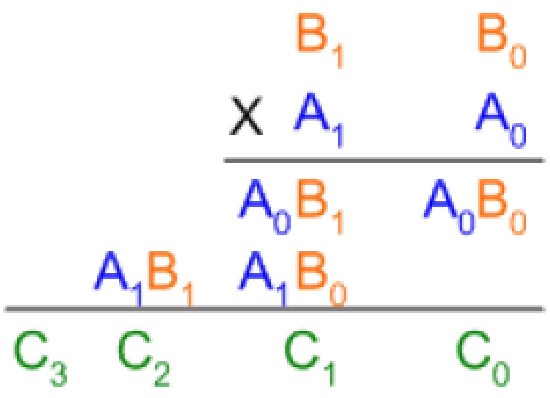
Multiplication of binary numbers is performed in the same way as with decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.

The product is obtained by adding these shifted partial products.

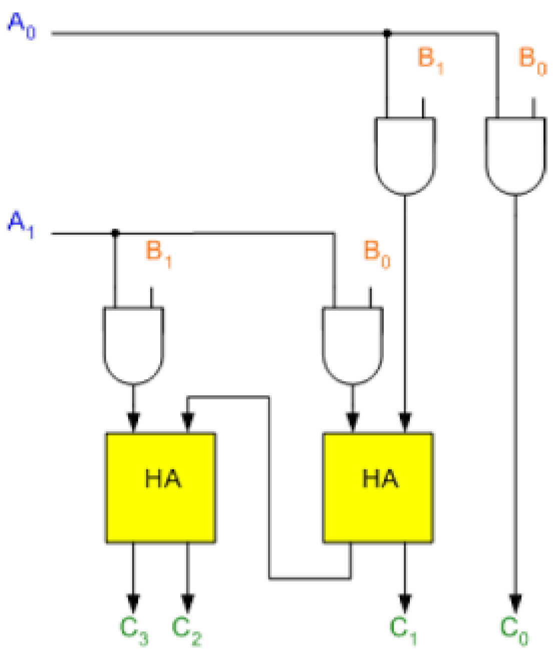
Consider an example of multiplication of two numbers, say A and B (2 bits each), C = A x B.

The first partial product is formed by multiplying the B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.

The second partial product is formed by multiplying the B_1B_0 by A_1 and is shifted one position to the left.



The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use Full Adders.

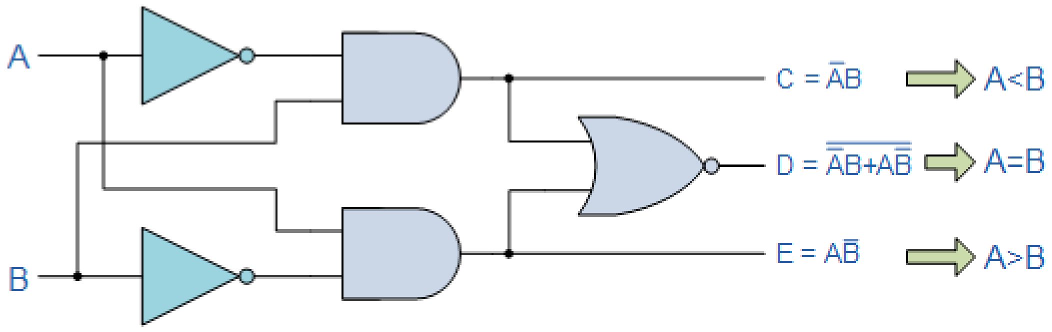


The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure. A binary multiplier with more bits can be constructed in a similar manner. Consider another example of multiplying two numbers, say A (3-bit number) and B (4-bit number). Each bit of A (the multiplier) is ANDed with each bit of B (the multicand) as shown in the Figure.

$$\begin{array}{ccccccc}
 & & & B_3 & B_2 & B_1 & B_0 \\
 & & & \times & A_2 & A_1 & A_0 \\
\cline{4-7}
 & & & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 \\
 & & A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & \\
 & A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & \\
\hline
C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}$$

The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the final product.

Since J = 3 and K = 4, 12 (J x K) AND gates and two 4-bit ((J - 1) K-bit) adders are needed to produce a product of seven (J + K) bits. Its circuit is shown in the Figure.
Note that 0 is applied at the most significant bit of augend of first 4-bit adder because the least significant bit of the product does not have to go through an adder



Digital Comparator

Another common and very useful combinational logic circuit is that of the **Digital Comparator** circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals present at their input terminals and produce an output depending upon the condition of those inputs.

For example, along with being able to add and subtract binary numbers we need to be able to compare them and determine whether the value of input A is greater than, smaller than or equal to the value at input B etc. The digital comparator accomplishes this using several logic gates that operate on the principles of *Boolean Algebra*. There are two main types of **Digital Comparator** available and these are.

- 1. Identity Comparator – an *Identity Comparator* is a digital comparator that has only one output terminal for when A = B either "HIGH" A = B = 1 or "LOW" A = B = 0
- 2. Magnitude Comparator – a *Magnitude Comparator* is a digital comparator which has three output terminals, one each for equality, A = B greater than, A > B and less than A < B

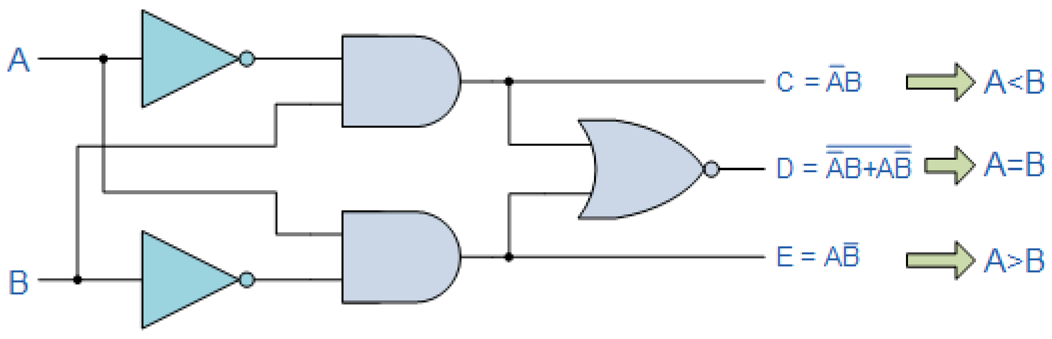
The purpose of a **Digital Comparator** is to compare a set of variables or unknown numbers, for example A (A1, A2, A3, …. An, etc) against that of a constant or unknown value such as B (B1, B2, B3, …. Bn, etc) and produce an output condition or flag depending upon the result of the comparison. For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

$$A > B,\quad A = B,\quad A < B$$

Which means: A is greater than B, A is equal to B, and A is less than B

This is useful if we want to compare two variables and want to produce an output when any of the above three conditions are achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

1-bit Digital Comparator Circuit



Then the operation of a 1-bit digital comparator is given in the following Truth Table.

Digital Comparator Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

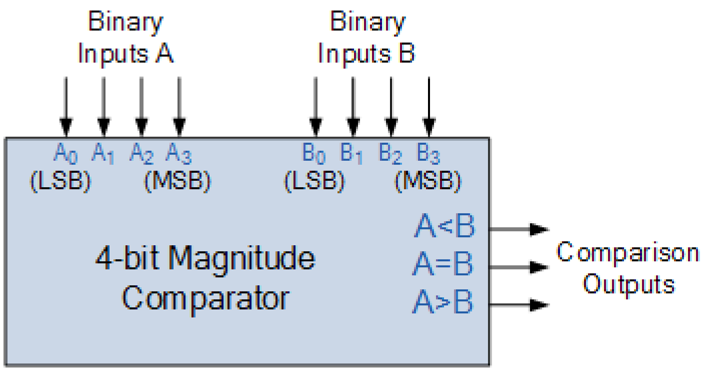
You may notice two distinct features about the comparator from the above truth table. Firstly, the circuit does not distinguish between either two "0" or two "1"s as an output A = B is produced when they are both equal, either A = B = "0" or A = B = "1". Secondly, the output condition for A = B resembles that of a commonly available logic gate, the Exclusive-NOR or Ex-NOR function (equivalence) on each of the n-bits giving: Q = A ⊕ B

Digital comparators actually use Exclusive-NOR gates within their design for comparing their respective pairs of bits. When we are comparing two binary or BCD values or variables against each other, we are comparing the "magnitude" of these values, a logic "0" against a logic "1" which is where the term Magnitude Comparator comes from.

As well as comparing individual bits, we can design larger bit comparators by cascading together n of these and produce a n-bit comparator just as we did for the n-bit adder in the previous tutorial. Multi-bit comparators can be constructed to compare whole binary or BCD words to produce an output if one word is larger, equal to or less than the other.

A very good example of this is the 4-bit Magnitude Comparator. Here, two 4-bit words ("nibbles") are compared to each other to produce the relevant output with one word connected to inputs A and the other to be compared against connected to input B as shown below.

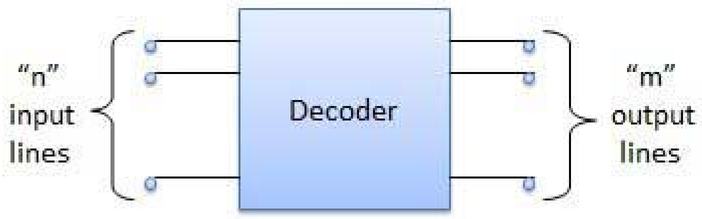
4-bit Magnitude Comparator



Decoder

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder.

Block diagram



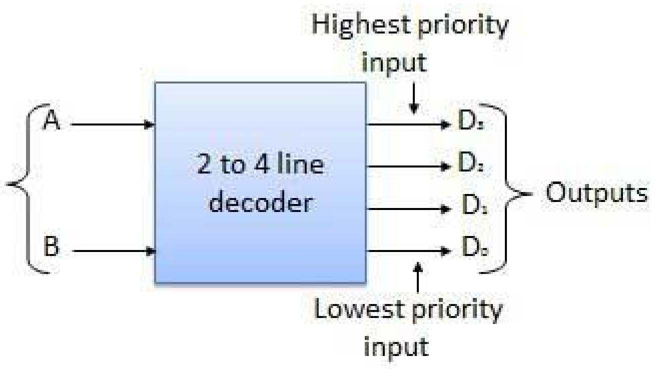
Examples of Decoders are following.

- Code converters
- BCD to seven segment decoders
- Nixie tube decoders
- Relay actuator

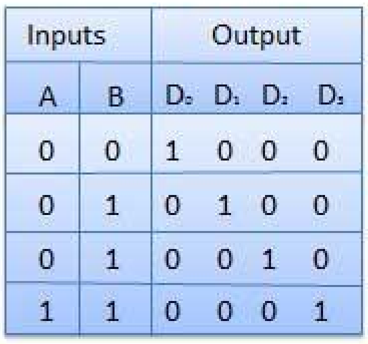
2 to 4 Line Decoder

The block diagram of 2 to 4 line decoder is shown in the fig. A and B are the two inputs where D through D are the four outputs. Truth table explains the operations of a decoder. It shows that each output is 1 for only a specific combination of inputs.

Block diagram

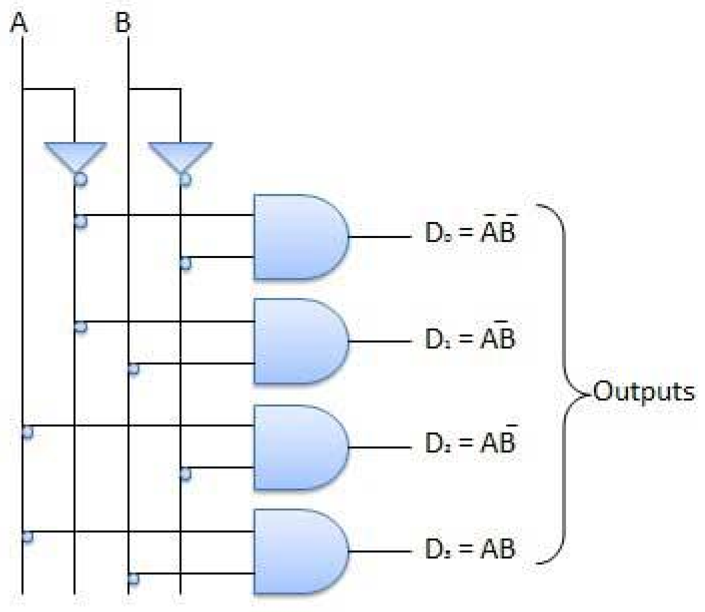


Truth Table



Inputs		Output			
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
0	1	0	0	1	0
1	1	0	0	0	1

Logic Circuit



in this topic we will try to discuss about Combinational Logic Implementation full adder circuit with a decoder and two OR Gates.

To do this at first we need to review the truth table of Full Adder circuit.

Inputs			Outputs	
a	b	c	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0

1	1	0	1	0
1	1	1	1	1

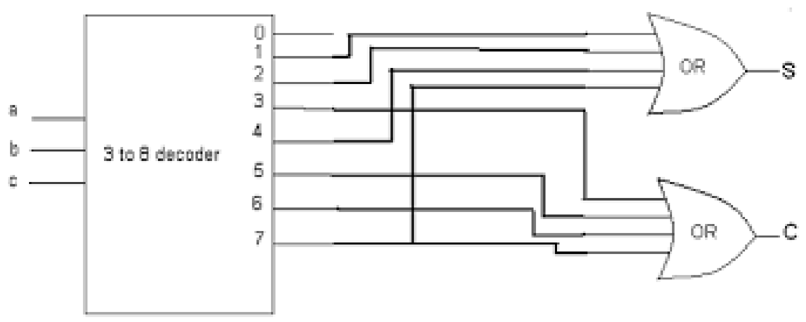
Truth table of Full Adder Circuit

From the truth table we have been found that
S(a,b,c)=sum(1,2,4,7).,C(a,b,c)=sum(3,5,6,7)

As there are three inputs and eight min-terms, so have to use 3 to 8 line decorder. The generates the eight min-terms for a, b, c.

The OR Gate for output S forms the sum of min-terms 1, 2, 4 and 7. The OR Gate output C forms the sum of min-terms 3, 5, 6 and 7.

Circuit Diagram



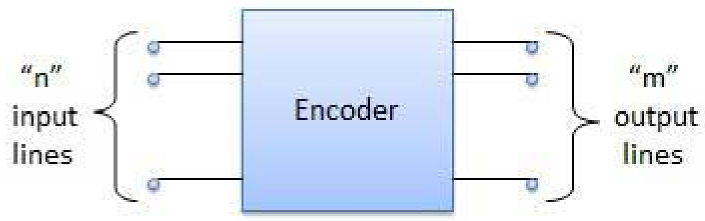
A function with a long list of min-terms requires an OR Gate with a large number of inputs. A function F having a list of K min-terms can be expressed in its complemented form F' with 2^n-K min-terms. If the number of min-terms in a function is greater than 2^n/2, then F' can be expressed with fewer min-terms than required for F. In such a case, it is suitable to use a NOR Gate to sum the min-terms of F'. The output of the NOR Gate will generate the normal output F.

The decoder method can be used to implement any combinational circuit. It is necessary to implementing with comparing to all other possible implementations to determine the solution.

Encoder

Encoder is a combinational circuit which is designed to perform the inverse operation of the decoder. An encoder has n number of input lines and m number of output lines. An encoder produces an m bit binary code corresponding to the digital input number. The encoder accepts an n input digital word and converts it into an m bit another digital word.

Block diagram



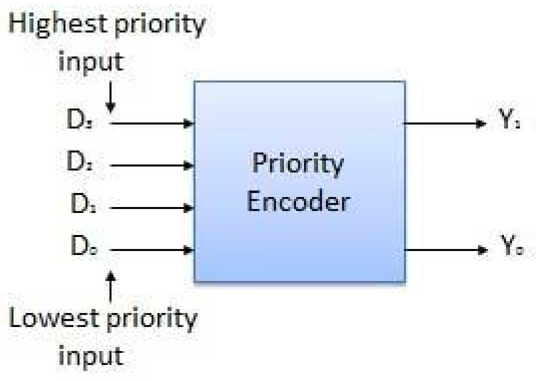
Examples of Encoders are following.

- Priority encoders
- Decimal to BCD encoder
- Octal to binary encoder
- Hexadecimal to binary encoder

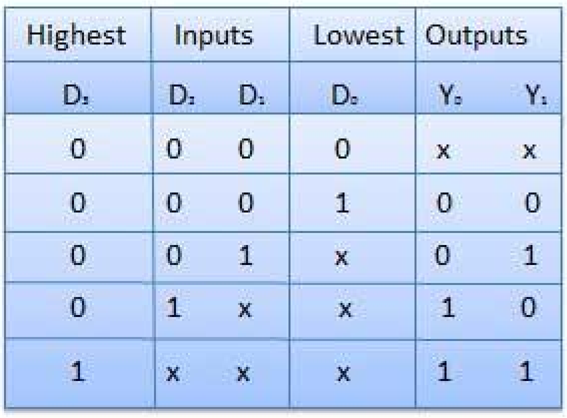
Priority Encoder

This is a special type of encoder. Priority is given to the input lines. If two or more input line are 1 at the same time, then the input line with highest priority will be considered. There are four input D_0, D_1, D_2, D_3 and two output Y_0, Y_1. Out of the four input D_3 has the highest priority and D_0 has the lowest priority. That means if $D_3 = 1$ then Y_1 Y_1 = 11 irrespective of the other inputs. Similarly if $D_3 = 0$ and $D_2 = 1$ then Y_1 Y_0 = 10 irrespective of the other inputs.

Block diagram

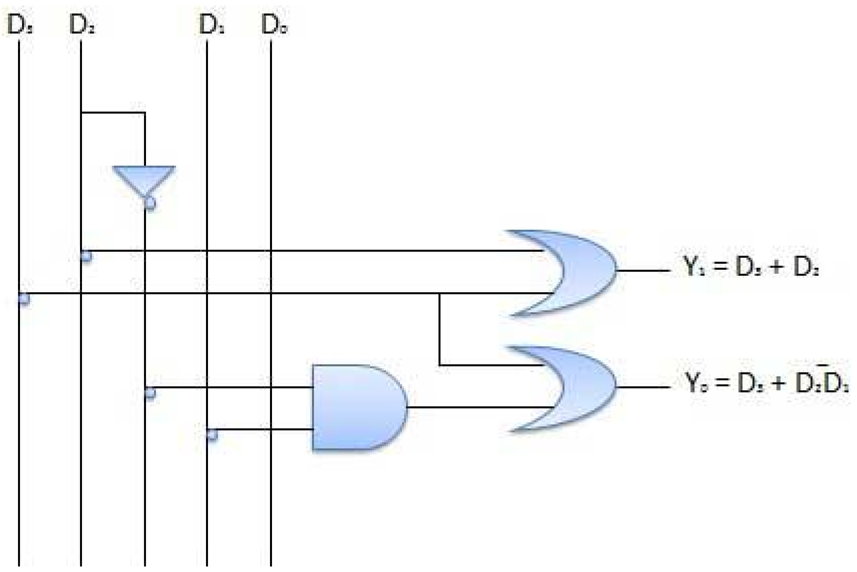


Truth Table



Highest	Inputs		Lowest	Outputs	
D_3	D_2	D_1	D_0	Y_0	Y_1
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

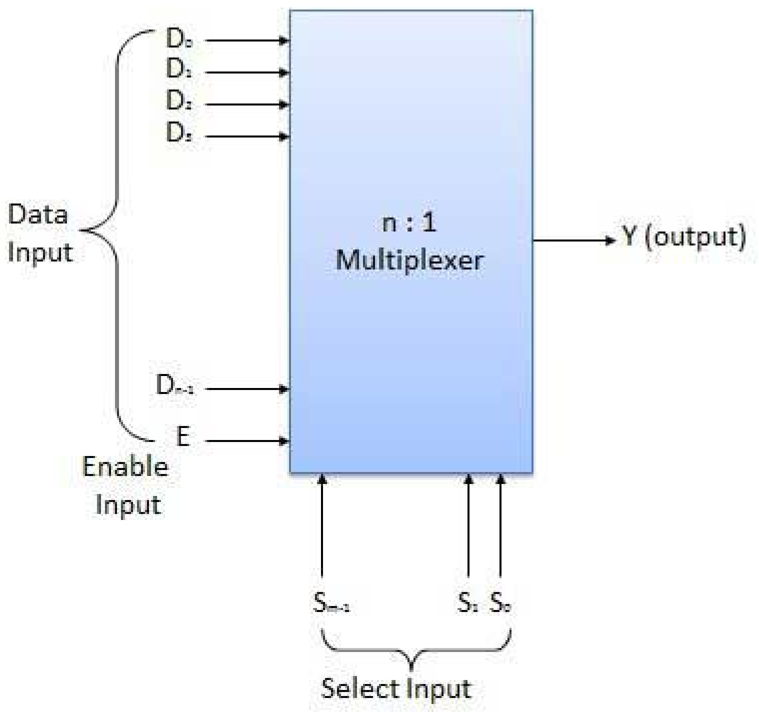
Logic Circuit



Multiplexers

Multiplexer is a special type of combinational circuit. There are n-data inputs, one output and m select inputs with 2m = n. It is a digital circuit which selects one of the n data inputs and routes it to the output. The selection of one of the n inputs is done by the selected inputs. Depending on the digital code applied at the selected inputs, one out of n data sources is selected and transmitted to the single output Y. E is called the strobe or enable input which is useful for the cascading. It is generally an active low terminal that means it will perform the required operation when it is low.

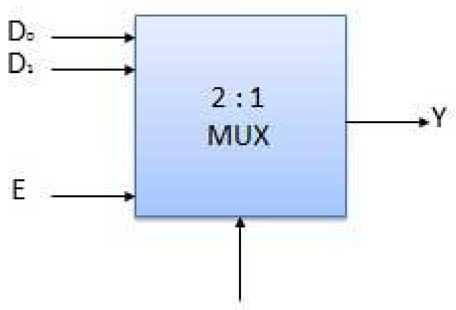
Block diagram



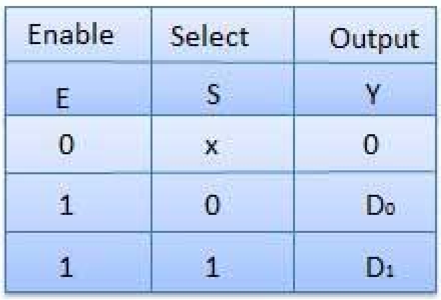
Multiplexers come in multiple variations

- 2 : 1 multiplexer
- 4 : 1 multiplexer
- 16 : 1 multiplexer
- 32 : 1 multiplexer

Block Diagram



Truth Table



Enable	Select	Output
E	S	Y
0	x	0
1	0	D_0
1	1	D_1

x = Don't care

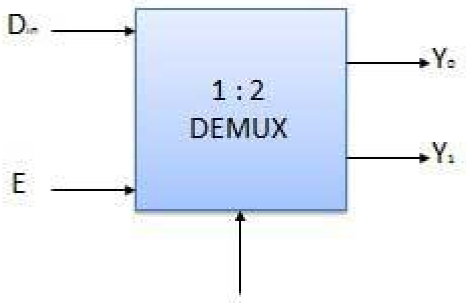
Demultiplexers

A demultiplexer performs the reverse operation of a multiplexer i.e. it receives one input and distributes it over several outputs. It has only one input, n outputs, m select input. At a time only one output line is selected by the select lines and the input is transmitted to the selected output line. A de-multiplexer is equivalent to a single pole multiple way switch as shown in fig.

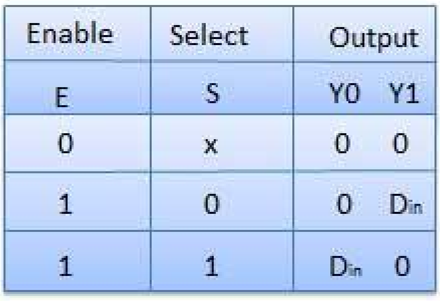
Demultiplexers come in multiple variations.

- 1 : 2 demultiplexer
- 1 : 4 demultiplexer
- 1 : 16 demultiplexer
- 1 : 32 demultiplexer

Block diagram



Truth Table



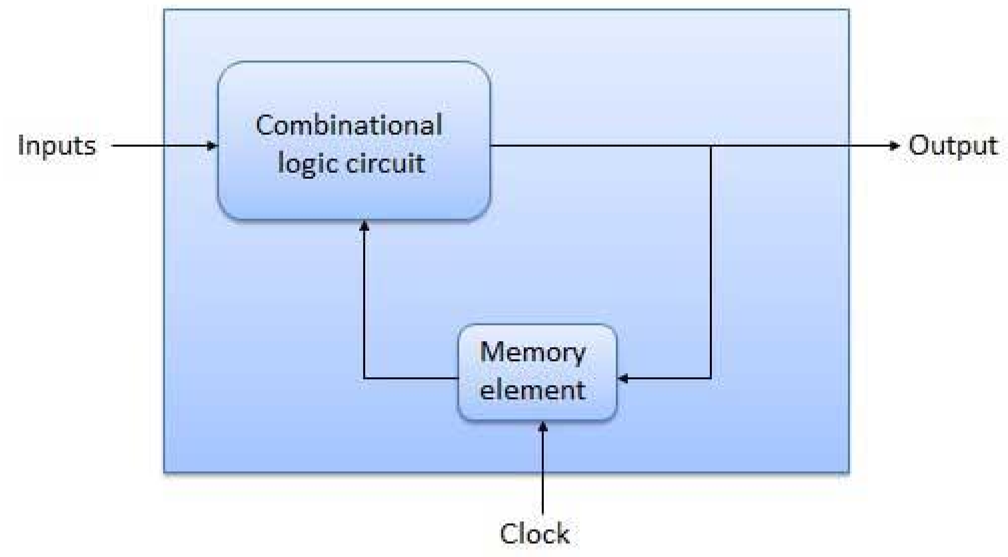
Enable	Select	Output	
E	S	Y0	Y1
0	x	0	0
1	0	0	D_{in}
1	1	D_{in}	0

x = Don’t care

UNIT IV
SEQUENTIAL LOGIC CIRCUITS:

The combinational circuit does not use any memory. Hence the previous state of input does not have any effect on the present state of the circuit. But sequential circuit has memory so output can vary based on input. This type of circuits uses previous input, output, clock and a memory element.

Block diagram

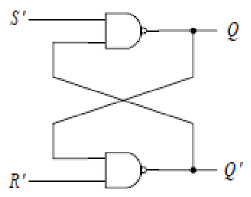


Flip Flop

Flip flop is a sequential circuit which generally samples its inputs and changes its outputs only at particular instants of time and not continuously. Flip flop is said to be edge sensitive or edge triggered rather than being level triggered like latches.

SR Latch

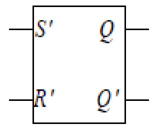
The bistable element is able to remember or store one bit of information. However, because it does not have any inputs, we cannot change the information bit that is stored in it. In order to change the information bit, we need to add inputs to the circuit. The simplest way to add inputs is to replace the two inverters with two NAND gates. This circuit is called a *SR latch*. In addition to the two outputs Q and Q', there are two inputs S' and R' for *set* and *reset* respectively. Following the convention, the prime in S and R denotes that these inputs are active low. The SR latch can be in one of two states: a set state when $Q = 1$, or a reset state when $Q = 0$.



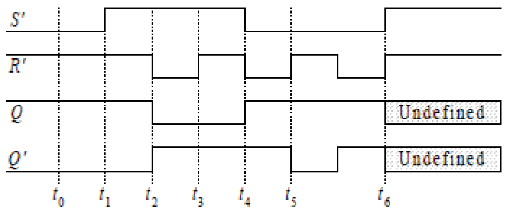
(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	×	1	1
0	1	×	1	0
1	0	×	0	1
1	1	0	0	1
1	1	1	1	0

(b)



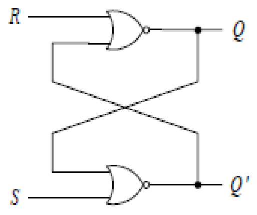
(c)



(d)

SR latch: (a) circuit using NAND gates; (b) truth table; (c) logic symbol; (d) timing diagram.

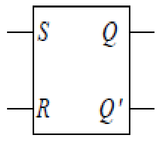
To make the SR latch go to the set state, we simply assert the *S'* input by setting it to 0. Remember that 0 NAND anything gives a 1, hence *Q* = 1 and the latch is set. If *R'* is not asserted (*R'* = 1), then the output of the bottom NAND gate will give a 0, and so *Q'* = 0. This situation is shown in Figure 4 (d) at time t_0. If we de-assert *S'* so that *S'* = *R'* = 1, the latch will remain at the set state because *Q'*, the second input to the top NAND gate, is 0 which will keep *Q* = 1 as shown at time t_1. At time t_2 we reset the latch by making *R'* = 0. Now, *Q'* goes to 1 and this will force *Q* to go to a 0. If we de-assert *R'* so that again we have *S'* = *R'* = 1, this time the latch will remain at the reset state as shown at time t_3. Notice the two times (at t_1 and t_3) when both *S'* and *R'* are de-asserted. At t_1, *Q* is at a 1, whereas, at t_3, *Q* is at a 0. When both inputs are de-asserted, the SR latch maintains its previous state. Previous to t1, Q has the value 1, so at t1, Q remains at a 1. Similarly, previous to t3, Q has the value 0, so at t3, Q remains at a 0.



(a)

S	R	Q	Q_{next}	Q_{next}'
0	0	0	0	1
0	0	1	1	0
0	1	×	0	1
1	0	×	1	0
1	1	×	0	0

(b)



(c)

re 5. SR latch: (a) circuit using NOR gates; (b) truth table; (c) logic symbol.

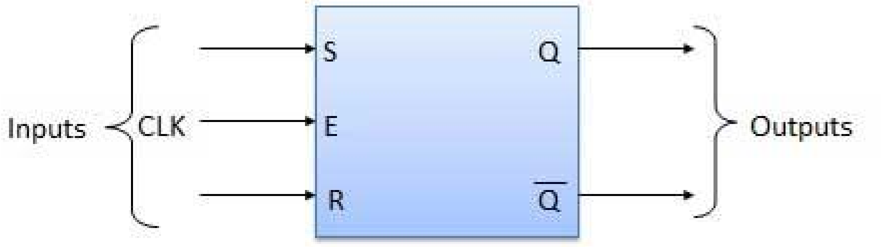
If both S' and R' are asserted, then both Q and Q' are equal to 1 as shown at time t4. If one of the input signals is de-asserted earlier than the other, the latch will end up in the state forced by the signal that was de-asserted later as shown at time t5. At t5, R' is de-asserted first, so the latch goes into the normal set state with Q = 1 and Q' = 0.

A problem exists if both S' and R' are de-asserted at exactly the same time as shown at time t6. If both gates have exactly the same delay then they will both output a 0 at exactly the same time. Feeding the zeros back to the gate input will produce a 1, again at exactly the same time, which again will produce a 0, and so on and on. This oscillating behavior, called the critical race, will continue forever. If the two gates do not have exactly the same delay then the situation is similar to de-asserting one input before the other, and so the latch will go into one state or the other. However, since we do not know which is the faster gate, therefore, we do not know which state the latch will go into. Thus, the latch's next state is undefined.

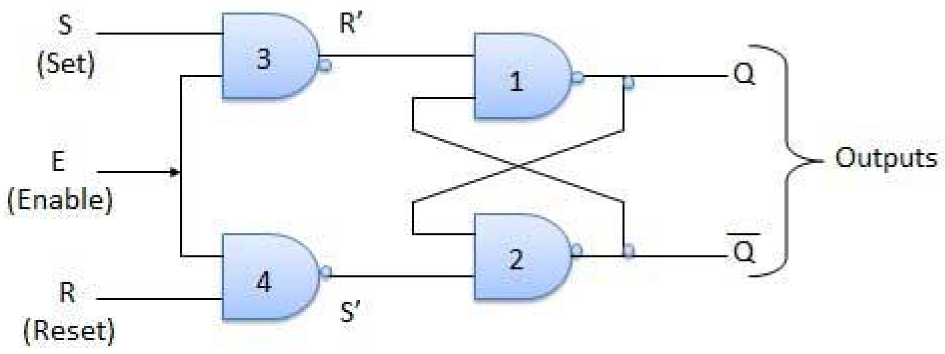
S-R Flip Flop

It is basically S-R latch using NAND gates with an additional **enable** input. It is also called as level triggered SR-FF. For this, circuit in output will take place if and only if the enable input (E) is made active. In short this circuit will operate as an S-R latch if E = 1 but there is no change in the output if E = 0.

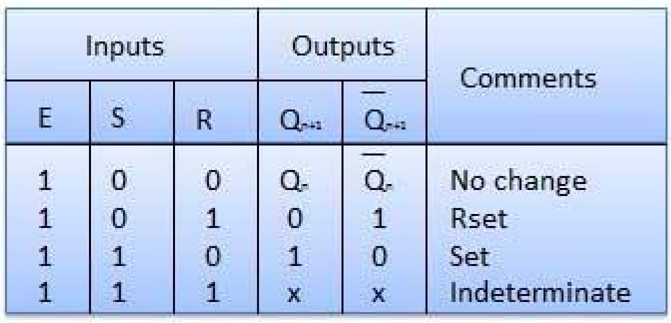
Block Diagram



Circuit Diagram



Truth Table



Inputs			Outputs		Comments
E	S	R	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	Q_n	$\overline{Q}_n$	No change
1	0	1	0	1	Rset
1	1	0	1	0	Set
1	1	1	x	x	Indeterminate

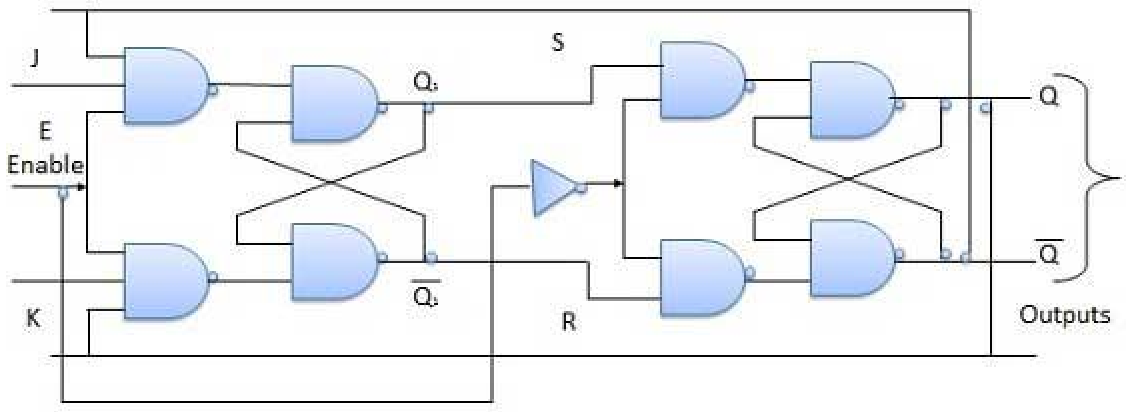
Operation

S.N.	Condition	Operation
1	**S = R = 0 : No change**	If S = R = 0 then output of NAND gates 3 and 4 are forced to become 1. Hence R' and S' both will be equal to 1. Since S' and R' are the input of the basic S-R latch using NAND gates, there will be no change in the state of outputs.
2	**S = 0, R = 1, E = 1**	Since S = 0, output of NAND-3 i.e. R' = 1 and E = 1 the output of NAND-4 i.e. S' = 0. Hence Q_{n+1} = 0 and Q_{n+1} bar = 1. This is reset condition.
3	**S = 1, R = 0, E = 1**	Output of NAND-3 i.e. R' = 0 and output of NAND-4 i.e. S' = 1. Hence output of S-R NAND latch is Q_{n+1} = 1 and Q_{n+1} bar = 0. This is the reset condition.
4	**S = 1, R = 1, E = 1**	As S = 1, R = 1 and E = 1, the output of NAND gates 3 and 4 both are 0 i.e. S' = R' = 0. Hence the **Race** condition will occur in the basic NAND latch.

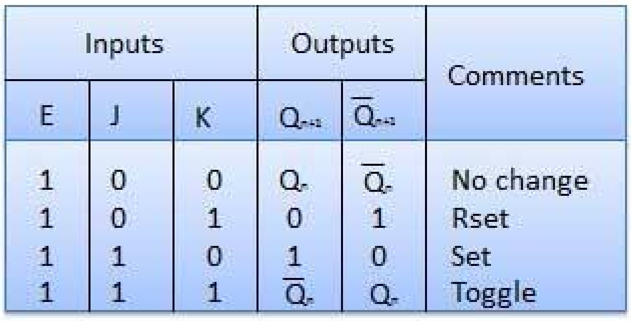
Master Slave JK Flip Flop

Master slave JK FF is a cascade of two S-R FF with feedback from the output of second to input of first. Master is a positive level triggered. But due to the presence of the inverter in the clock line, the slave will respond to the negative level. Hence when the clock = 1 (positive level) the master is active and the slave is inactive. Whereas when clock = 0 (low level) the slave is active and master is inactive.

Circuit Diagram



Truth Table



Inputs			Outputs		Comments
E	J	K	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	Q_n	$\overline{Q}_n$	No change
1	0	1	0	1	Rset
1	1	0	1	0	Set
1	1	1	$\overline{Q}_n$	Q_n	Toggle

Operation

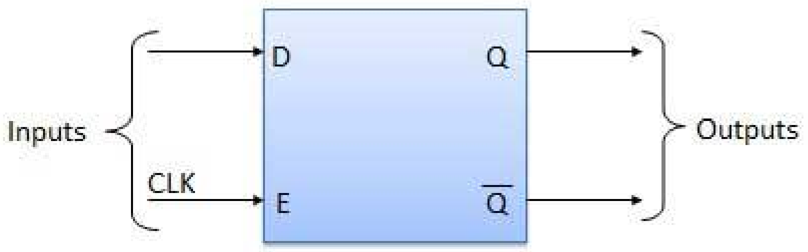
S.N.	Condition	Operation
1	**J = K = 0 (No change)**	When clock = 0, the slave becomes active and master is inactive. But since the S and R inputs have not changed, the slave outputs will also remain unchanged. Therefore outputs will not change if J = K =0.
2	**J = 0 and K = 1 (Reset)**	Clock = 1 − Master active, slave inactive. Therefore outputs of the master become Q_1 = 0 and Q_1 bar = 1. That means S = 0 and R =1. Clock = 0 − Slave active, master inactive. Therefore outputs of the slave become Q = 0 and Q bar = 1. Again clock = 1 − Master active, slave inactive. Therefore even with the changed outputs Q = 0 and Q bar = 1 fed back to master, its output will be Q1 = 0 and Q1 bar = 1. That means S = 0 and R = 1. Hence with clock = 0 and slave becoming active the outputs of slave will remain Q = 0 and Q bar = 1. Thus we get a stable output from the Master slave.
3	**J = 1 and K = 0 (Set)**	Clock = 1 − Master active, slave inactive. Therefore outputs of the master become Q_1 = 1 and Q_1 bar = 0. That means S = 1 and R =0. Clock = 0 − Slave active, master inactive. Therefore outputs of the slave become Q = 1 and Q bar = 0. Again clock = 1 − then it can be shown that the outputs of the slave are stabilized to Q = 1 and Q bar = 0.
4	**J = K = 1 (Toggle)**	Clock = 1 − Master active, slave inactive. Outputs of master will toggle. So S and R also will be inverted. Clock = 0 − Slave active, master inactive. Outputs of slave will toggle. These changed outputs are returned back to the master inputs. But since clock = 0, the master is still inactive. So it does not respond to these changed outputs.

This avoids the multiple toggling which leads to the race around condition. The master slave flip flop will avoid the race around condition.

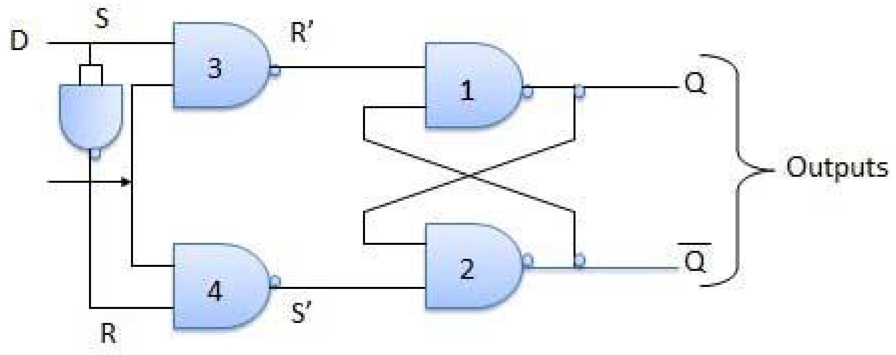
Delay Flip Flop / D Flip Flop

Delay Flip Flop or D Flip Flop is the simple gated S-R latch with a NAND inverter connected between S and R inputs. It has only one input. The input data is appearing at the output after some time. Due to this data delay between i/p and o/p, it is called delay flip flop. S and R will be the complements of each other due to NAND inverter. Hence S = R = 0 or S = R = 1, these input condition will never appear. This problem is avoid by SR = 00 and SR = 1 conditions.

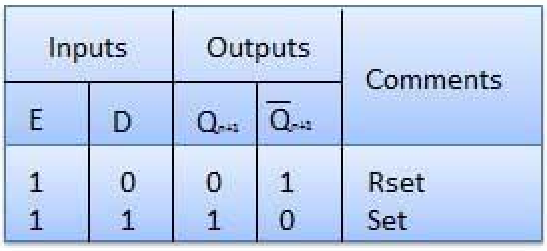
Block Diagram



Circuit Diagram



Truth Table



Inputs		Outputs		Comments
E	D	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	0	1	Rset
1	1	1	0	Set

Operation

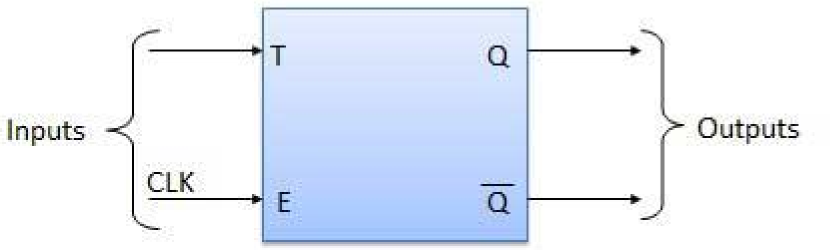
S.N.	Condition	Operation
1	**E = 0**	Latch is disabled. Hence no change in output.
2	**E = 1 and D = 0**	If E = 1 and D = 0 then S = 0 and R = 1. Hence irrespective of the present state, the next state is Q_{n+1} = 0 and Q_{n+1} bar = 1. This is the reset condition.
3	**E = 1 and D = 1**	If E = 1 and D = 1, then S = 1 and R = 0. This will set the latch and Q_{n+1} = 1 and Q_{n+1} bar =

0 irrespective of the present state.

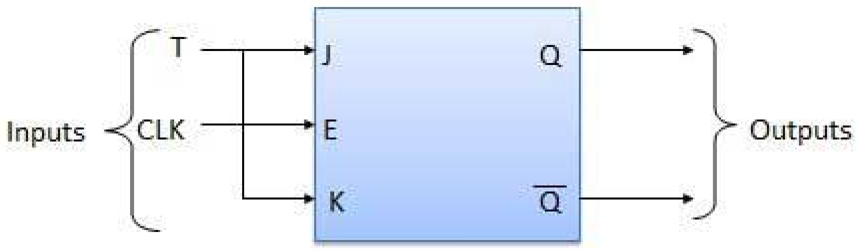
Toggle Flip Flop / T Flip Flop

Toggle flip flop is basically a JK flip flop with J and K terminals permanently connected together. It has only input denoted by **T** as shown in the Symbol Diagram. The symbol for positive edge triggered T flip flop is shown in the Block Diagram.

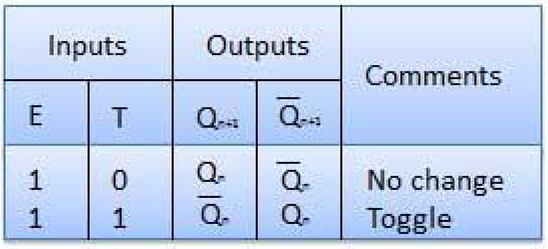
Symbol Diagram



Block Diagram



Truth Table



Inputs		Outputs		Comments
E	T	Q_{n+1}	$\overline{Q}_{n+1}$	
1	0	Q_n	$\overline{Q}_n$	No change
1	1	$\overline{Q}_n$	Q_n	Toggle

Operation

S.N.	Condition	Operation
1	**T = 0, J = K = 0**	The output Q and Q bar won't change
2	**T = 1, J = K = 1**	Output will toggle corresponding to every leading edge of clock signal.

REGISTER

Flip-flop is a 1 bit memory cell which can be used for storing the digital data. To increase the storage capacity in terms of number of bits, we have to use a group of flip-flop. Such a group of flip-flop is known as a **Register**. The **n-bit register** will consist of **n** number of flip-flop and it is capable of storing an **n-bit** word.

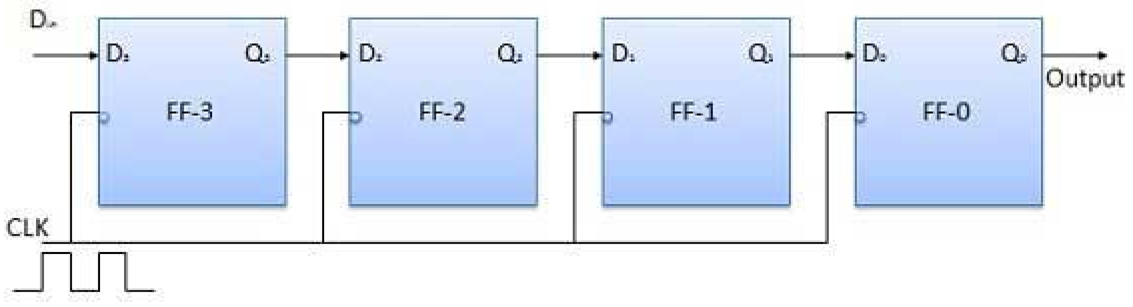
The binary data in a register can be moved within the register from one flip-flop to another. The registers that allow such data transfers are called as **shift registers**. There are four mode of operations of a shift register.

- Serial Input Serial Output
- Serial Input Parallel Output
- Parallel Input Serial Output
- Parallel Input Parallel Output

Serial Input Serial Output

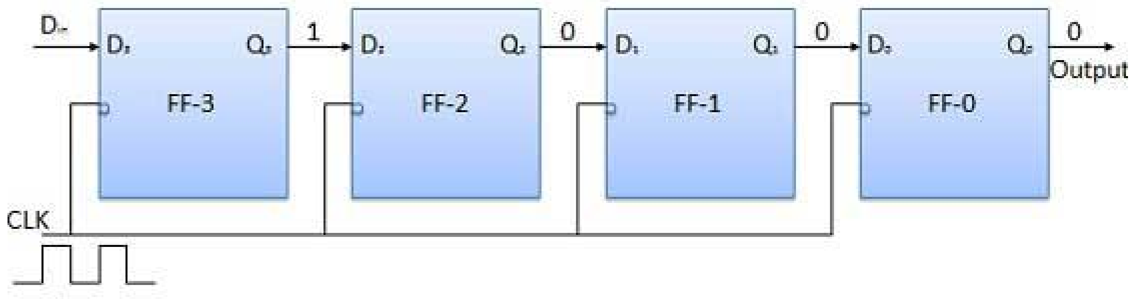
Let all the flip-flop be initially in the reset condition i.e. $Q_3 = Q_2 = Q_1 = Q_0 = 0$. If an entry of a four bit binary number 1 1 1 1 is made into the register, this number should be applied to $\mathbf{D_{in}}$ bit with the LSB bit applied first. The D input of FF-3 i.e. D_3 is connected to serial data input $\mathbf{D_{in}}$. Output of FF-3 i.e. Q_3 is connected to the input of the next flip-flop i.e. D_2 and so on.

Block Diagram

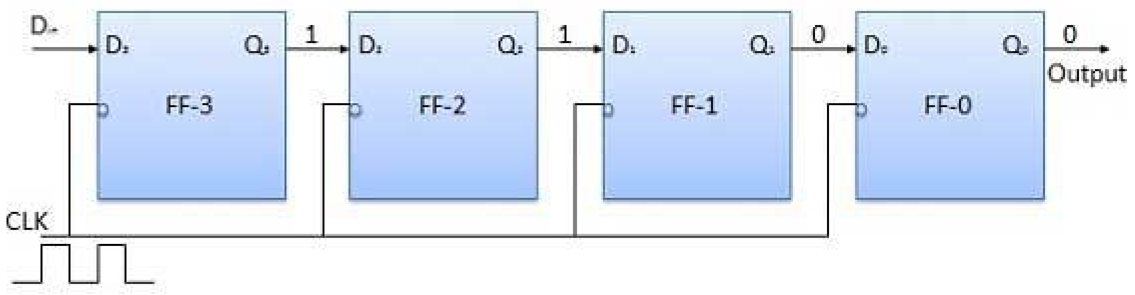


Operation

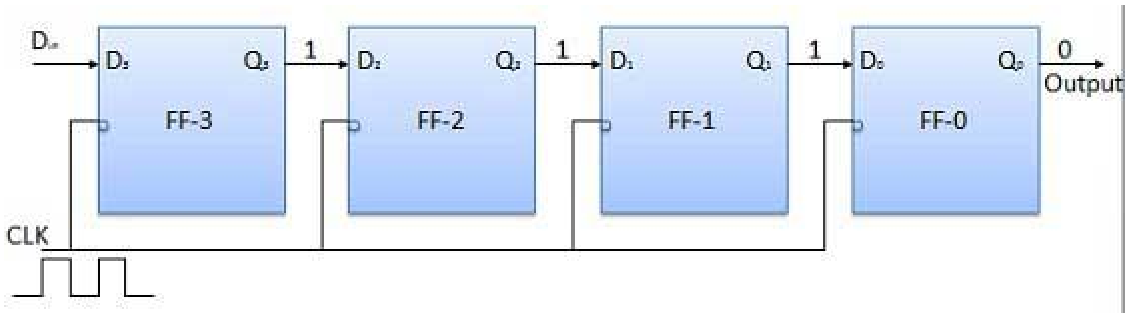
Before application of clock signal, let $Q_3\ Q_2\ Q_1\ Q_0$ = 0000 and apply LSB bit of the number to be entered to D_{in}. So $D_{in} = D_3 = 1$. Apply the clock. On the first falling edge of clock, the FF-3 is set, and stored word in the register is $Q_3\ Q_2\ Q_1\ Q_0$ = 1000.



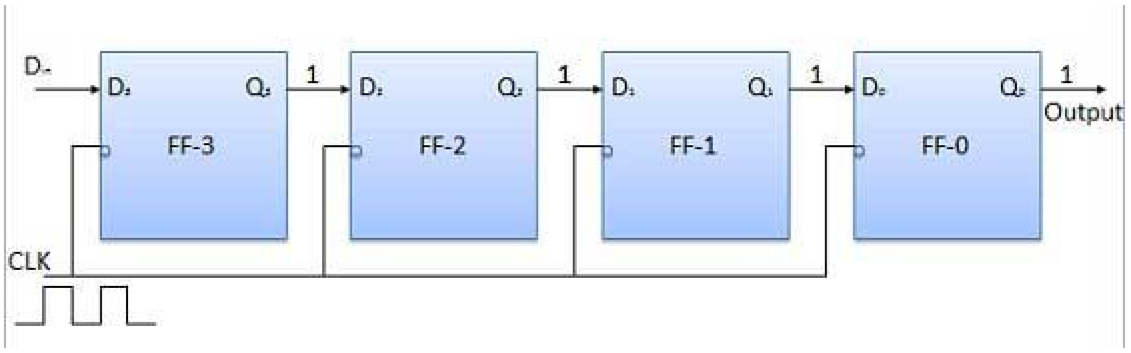
Apply the next bit to D_{in}. So D_{in} = 1. As soon as the next negative edge of the clock hits, FF-2 will set and the stored word change to $Q_3\ Q_2\ Q_1\ Q_0$ = 1100.



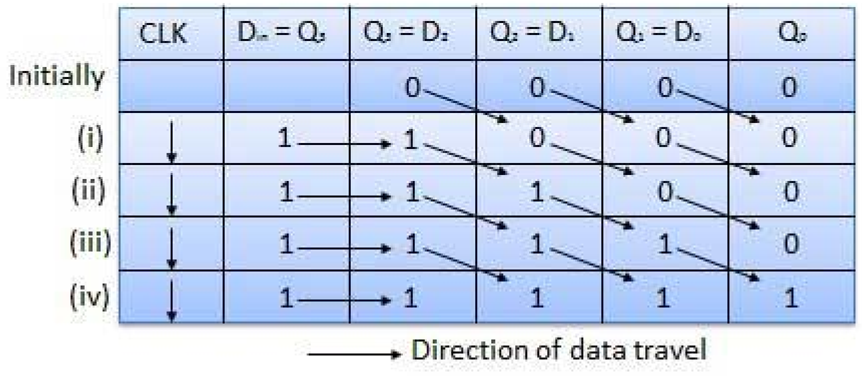
Apply the next bit to be stored i.e. 1 to D_{in}. Apply the clock pulse. As soon as the third negative clock edge hits, FF-1 will be set and output will be modified to $Q_3\ Q_2\ Q_1\ Q_0 = 1110$.



Similarly with $D_{in} = 1$ and with the fourth negative clock edge arriving, the stored word in the register is $Q_3\ Q_2\ Q_1\ Q_0 = 1111$.



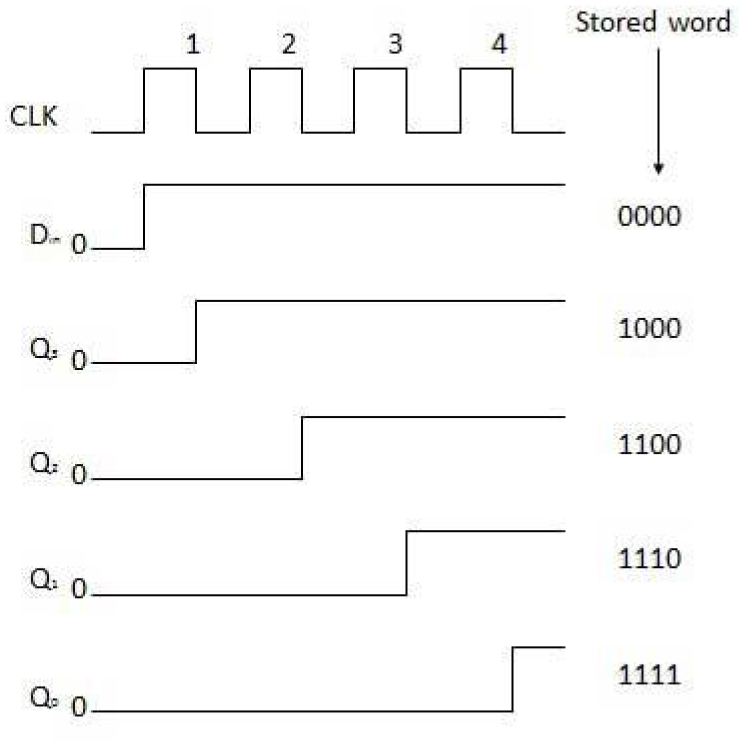
Truth Table



	CLK	$D_{in} = Q_3$	$Q_3 = D_2$	$Q_2 = D_1$	$Q_1 = D_0$	Q_0
Initially			0	0	0	0
(i)	↓	1	1	0	0	0
(ii)	↓	1	1	1	0	0
(iii)	↓	1	1	1	1	0
(iv)	↓	1	1	1	1	1

$\longrightarrow$ Direction of data travel

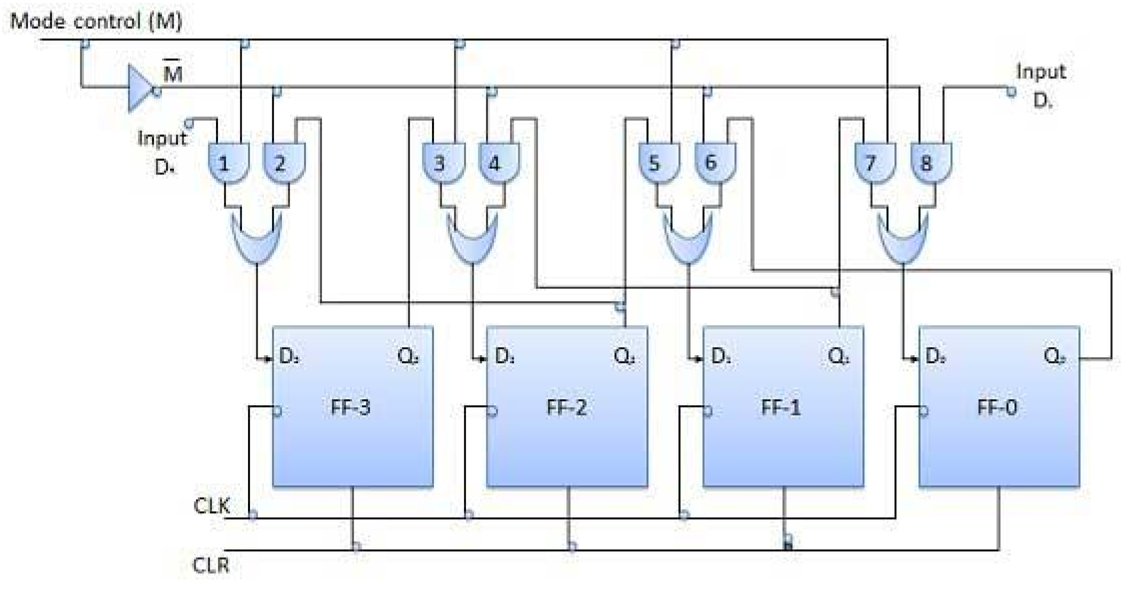
Waveforms



Bidirectional Shift Register

- If a binary number is shifted left by one position then it is equivalent to multiplying the original number by 2. Similarly if a binary number is shifted right by one position then it is equivalent to dividing the original number by 2.
- Hence if we want to use the shift register to multiply and divide the given binary number, then we should be able to move the data in either left or right direction.
- Such a register is called bi-directional register. A four bit bi-directional shift register is shown in fig.
- There are two serial inputs namely the serial right shift data input DR, and the serial left shift data input DL along with a mode select input (M).

Block Diagram



Operation

S.N.	Condition	Operation
1	**With M = 1 − Shift right operation**	If M = 1, then the AND gates 1, 3, 5 and 7 are enabled whereas the remaining AND gates 2, 4, 6 and 8 will be disabled. The data at D_R is shifted to right bit by bit from FF-3 to FF-0 on the application of clock pulses. Thus with M = 1 we get the serial right shift operation.
2	**With M = 0 − Shift left operation**	When the mode control M is connected to 0 then the AND gates 2, 4, 6 and 8 are enabled while 1, 3, 5 and 7 are disabled. The data at D_L is shifted left bit by bit from FF-0 to FF-3 on the application of clock pulses. Thus with M = 0 we get the serial right shift operation.

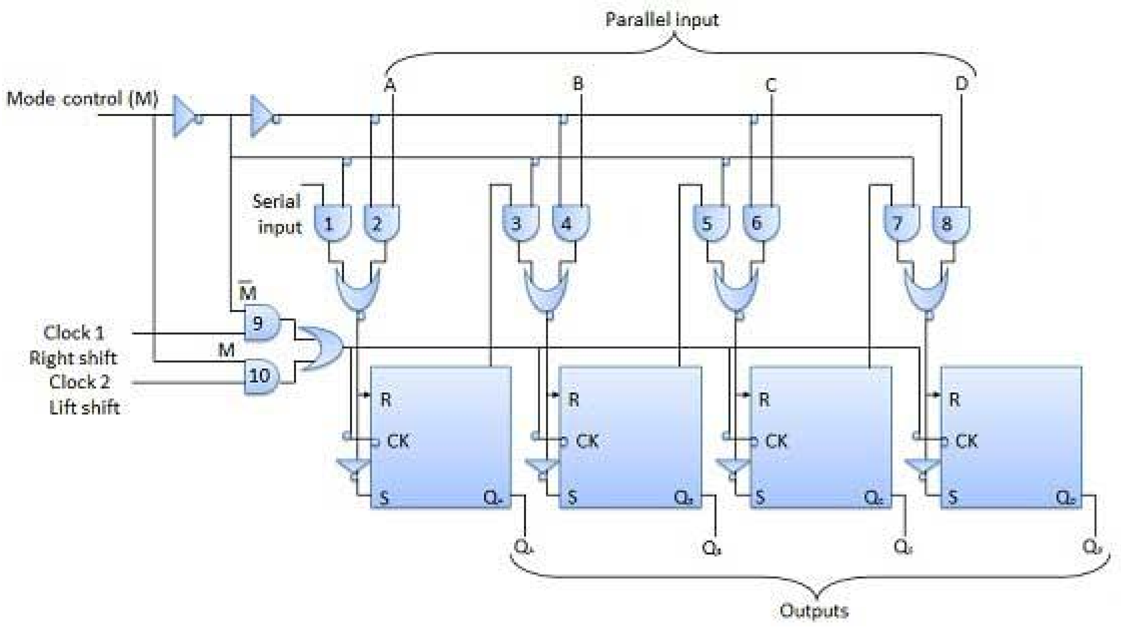
Universal Shift Register

A shift register which can shift the data in only one direction is called a uni-directional shift register. A shift register which can shift the data in both directions is called a bi-directional shift register. Applying the same logic, a shift register which can shift the data in both directions as well as load it parallely, is known as a universal shift register. The shift register is capable of performing the following operation −

- Parallel loading
- Lift shifting
- Right shifting

The mode control input is connected to logic 1 for parallel loading operation whereas it is connected to 0 for serial shifting. With mode control pin connected to ground, the universal shift register acts as a bi-directional register. For serial left operation, the input is applied to the serial input which goes to AND gate-1 shown in figure. Whereas for the shift right operation, the serial input is applied to D input.

Block Diagram



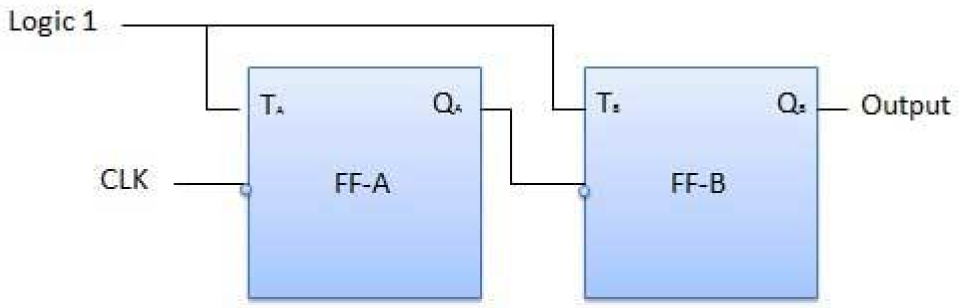
Counter is a sequential circuit. A digital circuit which is used for a counting pulses is known counter. Counter is the widest application of flip-flops. It is a group of flip-flops with a clock signal applied. Counters are of two types.

- Asynchronous or ripple counters.
- Synchronous counters.

Asynchronous or ripple counters

The logic diagram of a 2-bit ripple up counter is shown in figure. The toggle (T) flip-flop are being used. But we can use the JK flip-flop also with J and K connected permanently to logic 1. External clock is applied to the clock input of flip-flop A and Q_A output is applied to the clock input of the next flip-flop i.e. FF-B.

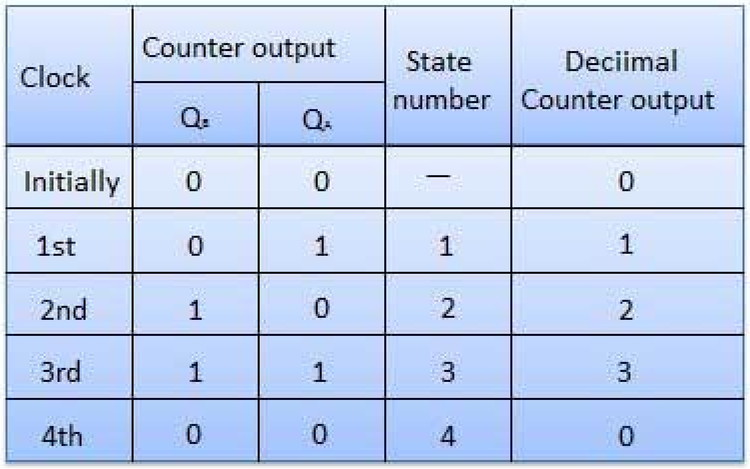
Logical Diagram



Operation

S.N.	Condition	Operation
1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will be equal to 1. Q_A is connected to clock input of FF-B. Since Q_A has changed from 0 to 1, it is treated as the positive clock edge by FF-B. There is no change in Q_B because FF-B is a negative edge triggered FF. Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A toggles again and Q_A = 0. The change in Q_A acts as a negative clock edge for FF-B. So it will also toggle, and Q_B will be 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On the arrival of 3rd negative clock edge, FF-A toggles again and Q_A become 1 from 0. Since this is a positive going change, FF-B does not respond to it and remains inactive. So Q_B does not change and continues to be equal to 1. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On the arrival of 4th negative clock edge, FF-A toggles again and Q_A becomes 1 from 0. This negative change in Q_A acts as clock pulse for FF-B. Hence it toggles to change Q_B from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

Truth Table



Clock	Counter output		State number	Deciimal Counter output
	Q_B	Q_A		
Initially	0	0	—	0
1st	0	1	1	1
2nd	1	0	2	2
3rd	1	1	3	3
4th	0	0	4	0

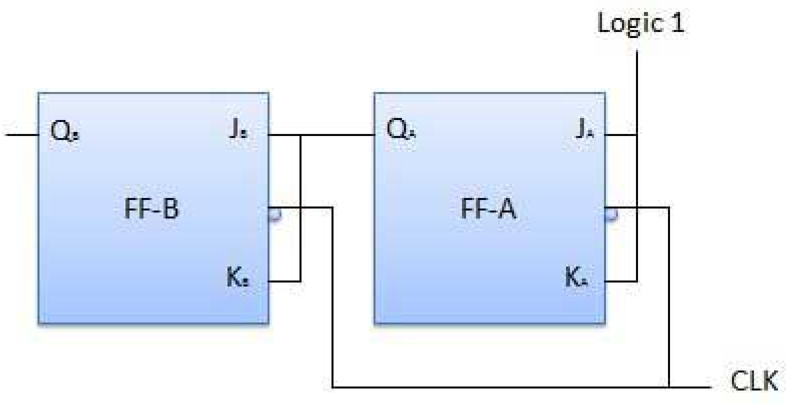
Synchronous counters

If the "clock" pulses are applied to all the flip-flops in a counter simultaneously, then such a counter is called as synchronous counter.

2-bit Synchronous up counter

The J_A and K_A inputs of FF-A are tied to logic 1. So FF-A will work as a toggle flip-flop. The J_B and K_B inputs are connected to Q_A.

Logical Diagram



Operation

S.N.	Condition	Operation
1	**Initially let both the FFs be in the reset state**	Q_BQ_A = 00 initially.
2	**After 1st negative clock edge**	As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will change from 0 to 1. But at the instant of application of negative clock edge, Q_A , $J_B = K_B = 0$. Hence FF-B will not change its state. So Q_B will remain 0. Q_BQ_A = 01 after the first clock pulse.
3	**After 2nd negative clock edge**	On the arrival of second negative clock edge, FF-A

		toggles again and Q_A changes from 1 to 0. But at this instant Q_A was 1. So J_B = K_B= 1 and FF-B will toggle. Hence Q_B changes from 0 to 1. Q_BQ_A = 10 after the second clock pulse.
4	**After 3rd negative clock edge**	On application of the third falling clock edge, FF-A will toggle from 0 to 1 but there is no change of state for FF-B. Q_BQ_A = 11 after the third clock pulse.
5	**After 4th negative clock edge**	On application of the next clock pulse, Q_A will change from 1 to 0 as Q_B will also change from 1 to 0. Q_BQ_A = 00 after the fourth clock pulse.

Classification of counters

Depending on the way in which the counting progresses, the synchronous or asynchronous counters are classified as follows −

- Up counters
- Down counters
- Up/Down counters

UP/DOWN Counter

Up counter and down counter is combined together to obtain an UP/DOWN counter. A mode control (M) input is also provided to select either up or down mode. A combinational circuit is required to be designed and used between each pair of flip-flop in order to achieve the up/down operation.

- Type of up/down counters
- UP/DOWN ripple counters
- UP/DOWN synchronous counter

UP/DOWN Ripple Counters

In the UP/DOWN ripple counter all the FFs operate in the toggle mode. So either T flip-flops or JK flip-flops are to be used. The LSB flip-flop receives clock directly. But the clock to every other FF is obtained from (Q = Q bar) output of the previous FF.

- **UP counting mode (M=0)** − The Q output of the preceding FF is connected to the clock of the next stage if up counting is to be achieved. For this mode, the mode select input M is at logic 0 (M=0).

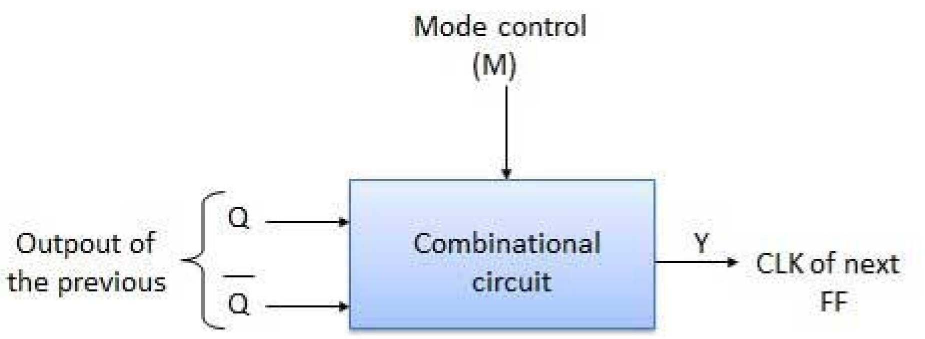
- **DOWN counting mode (M=1)** − If M = 1, then the Q bar output of the preceding FF is connected to the next FF. This will operate the counter in the counting mode.

Example

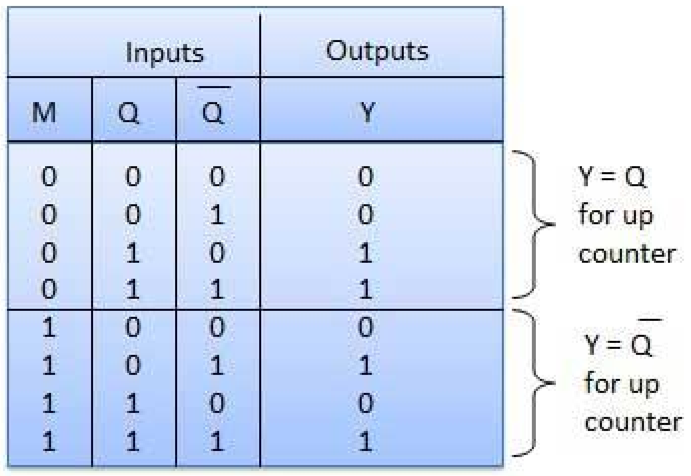
3-bit binary up/down ripple counter.

- 3-bit − hence three FFs are required.
- UP/DOWN − So a mode control input is essential.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple up counter, the Q output of preceding FF is connected to the clock input of the next one.
- For a ripple down counter, the Q bar output of preceding FF is connected to the clock input of the next one.
- Let the selection of Q and Q bar output of the preceding FF be controlled by the mode control input M such that, If M = 0, UP counting. So connect Q to CLK. If M = 1, DOWN counting. So connect Q bar to CLK.

Block Diagram



Truth Table



Inputs			Outputs
M	Q	$\overline{Q}$	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Y = Q for up counter

Y = $\overline{Q}$ for up counter

Operation

S.N.	Condition	Operation
1	**Case 1 − With M = 0 (Up counting mode)**	If M = 0 and M bar = 1, then the AND gates 1 and 3 in fig. will be enabled whereas the AND gates 2 and 4 will be disabled. Hence Q_A gets connected to the clock input of FF-B and Q_B gets connected to the clock input of FF-C. These connections are same as those for the normal up counter. Thus with M = 0 the circuit work as an up counter.
2	**Case 2: With M = 1 (Down counting mode)**	If M = 1, then AND gates 2 and 4 in fig. are enabled whereas the AND gates 1 and 3 are disabled. Hence Q_A bar gets connected to the clock input of FF-B and Q_B bar gets connected to the clock input of FF-C. These connections will produce a down counter. Thus with M = 1 the circuit works as a down counter.

Modulus Counter (MOD-N Counter)

The 2-bit ripple counter is called as MOD-4 counter and 3-bit ripple counter is called as MOD-8 counter. So in general, an n-bit ripple counter is called as modulo-N counter. Where, MOD number = 2^n.

Type of modulus

- 2-bit up or down (MOD-4)
- 3-bit up or down (MOD-8)
- 4-bit up or down (MOD-16)

Application of counters

- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator

UNIT-5 PROGRAMMABLE LOGIC & MEMORY

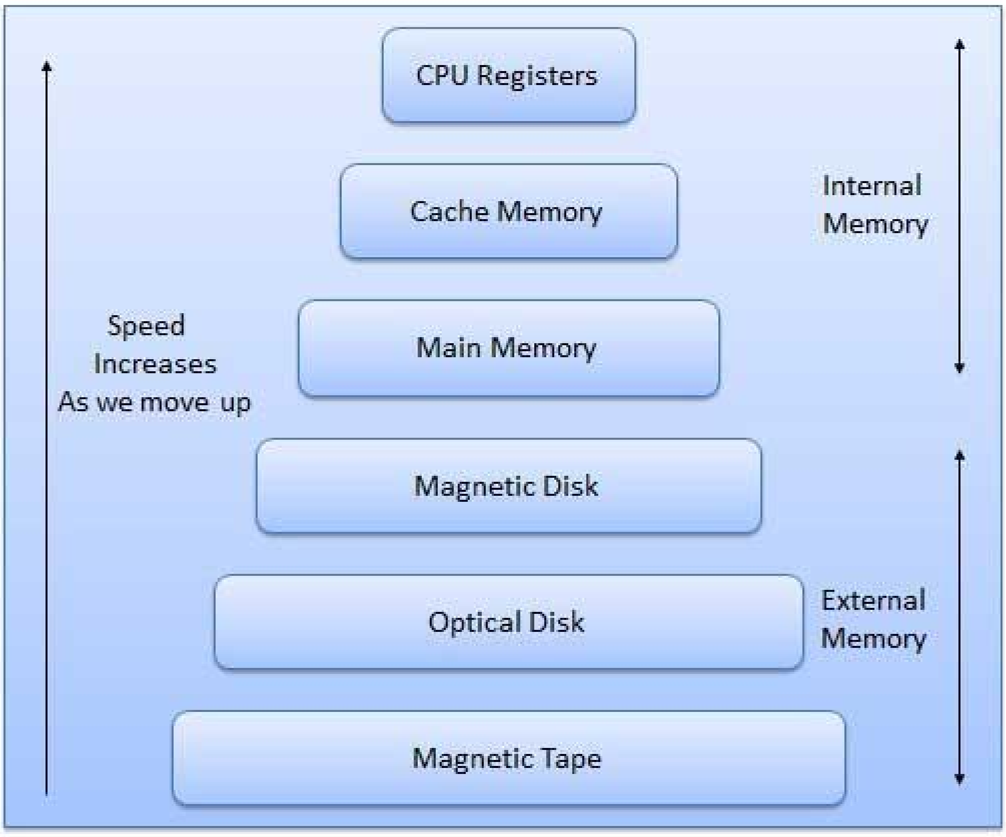
A memory is just like a human brain. It is used to store data and instruction. Computer memory is the storage space in computer where data is to be processed and instructions required for processing are stored.

The memory is divided into large number of small parts. Each part is called a cell. Each location or cell has a unique address which varies from zero to memory size minus one.

For example if computer has 64k words, then this memory unit has 64 * 1024 = 65536 memory location. The address of these locations varies from 0 to 65535.

Memory is primarily of two types

- **Internal Memory** – cache memory and primary/main memory
- **External Memory** – magnetic disk / optical disk etc.



Characteristics of Memory Hierarchy are following when we go from top to bottom.

- Capacity in terms of storage increases.
- Cost per bit of storage decreases.
- Frequency of access of the memory by the CPU decreases.
- Access time by the CPU increases.

RAM

A RAM constitutes the internal memory of the CPU for storing data, program and program result. It is read/write memory. It is called random access memory (RAM).

Since access time in RAM is independent of the address to the word that is, each storage location inside the memory is as easy to reach as other location & takes the same amount of time. We can reach into the memory at random & extremely fast but can also be quite expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup uninterruptible power system (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Static RAM (SRAM)

The word **static** indicates that the memory retains its contents as long as power remains applied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not have to be refreshed on a regular basis.

Because of the extra space in the matrix, SRAM uses more chips than DRAM for the same amount of storage space, thus making the manufacturing costs higher.

Static RAM is used as cache memory needs to be very fast and small.

Dynamic RAM (DRAM)

DRAM, unlike SRAM, must be continually **refreshed** in order for it to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory because it is cheap and small. All DRAMs are made up of memory cells. These cells are composed of one capacitor and one transistor.

ROM

ROM stands for Read Only Memory. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture.

A ROM, stores such instruction as are required to start computer when electricity is first turned on, this operation is referred to as bootstrap. ROM chip are not only used in the computer but also in other electronic items like washing machine and microwave oven.

Following are the various types of ROM –

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs. It is inexpensive ROM.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM programmer. Inside the PROM chip there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

The EPROM can be erased by exposing it to ultra-violet light for a duration of upto 40 minutes. Usually, an EPROM eraser achieves this function. During programming an electrical charge is trapped in an insulated gate region. The charge is retained for more than ten years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable and Programmable Read Only Memory)

The EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of re-programming is flexible but slow.

Serial Access Memory

Sequential access means the system must search the storage device from the beginning of the memory address until it finds the required piece of data. Memory device which supports such access is called a Sequential Access Memory or Serial Access Memory. Magnetic tape is an example of serial access memory.

Direct Access Memory

Direct access memory or Random Access Memory, refers to conditions in which a system can go directly to the information that the user wants. Memory device which supports such access is called a Direct Access Memory. Magnetic disks, optical disks are examples of direct access memory.

Cache Memory

Cache memory is a very high speed semiconductor memory which can speed up CPU. It acts as a buffer between the CPU and main memory. It is used to hold those parts of data and program which are most frequently used by CPU. The parts of data and programs, are transferred from disk to cache memory by operating system, from where CPU can access them.

Advantages

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

- Cache memory has limited capacity.
- It is very expensive.

Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory.

This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.

- Certain options and features of a program may be used rarely.

- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

- The ability to execute a program that is only partially in memory would counter many benefits.

- Less number of I/O would be needed to load or swap each user program into memory.

- A program would no longer be constrained by the amount of physical memory that is available.

- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Auxiliary Memory

Auxiliary memory is much larger in size than main memory but is slower. It normally stores system programs, instruction and data files. It is also known as secondary memory. It can also be used as an overflow/virtual memory in case the main memory capacity has been exceeded. Secondary memories cannot be accessed directly by a processor. First the data/information of auxiliary memory is transferred to the main memory and then that information can be accessed by the CPU. Characteristics of Auxiliary Memory are following −

- **Non-volatile memory** − Data is not lost when power is cut off.

- **Reusable** − The data stays in the secondary storage on permanent basis until it is not overwritten or deleted by the user.

- **Reliable** − Data in secondary storage is safe because of high physical stability of secondary storage device.
- **Convenience** − With the help of a computer software, authorised people can locate and access the data quickly.
- **Capacity** − Secondary storage can store large volumes of data in sets of multiple disks.
- **Cost** − It is much lesser expensive to store data on a tape or disk than primary memory.

The next three combinational components we will study are: ROM, PLA, and PAL.

ROM's PLA's and PAL's are storage Components. This might seem like a contradiction because earlier we said that combinational components don't have memory. This apparent contradiction results from an incomplete definition of the term combination component. A more precise definition is: a combinational component is a circuit that doesn't have memory *of past inputs*. (The outputs of a combinational component are completely determined by the current inputs.) The data in the storage components we are about to study are stored at design time before the component is added to a circuit. The data stored in sequential circuits comes from the inputs that are received while the component is active in the circuit.

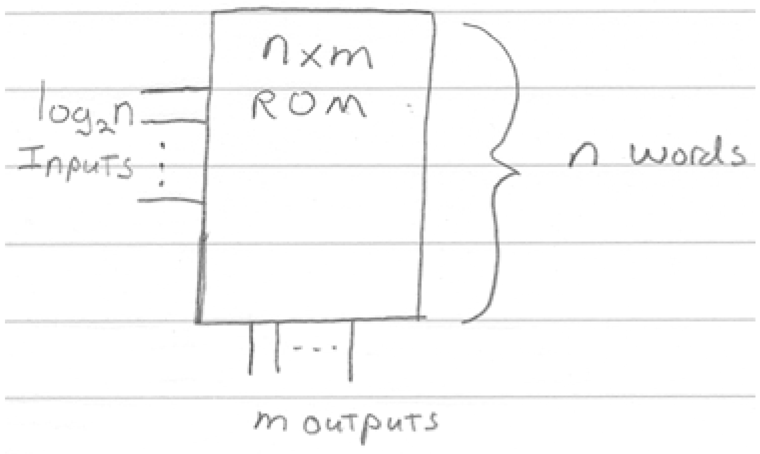
The storage components that are the topic of this lecture can be used to implement Boolean functions. This may also be the most efficient implementation. If the function is moderately complex a SSI implementation (individual gates) may be expensive in terms of the number of gates that have to be purchased and in terms of the number of connections required to wire them together. When a function is implemented inside a ROM, PLA, or PAL there is only one IC to purchase and fewer connections.

ROM

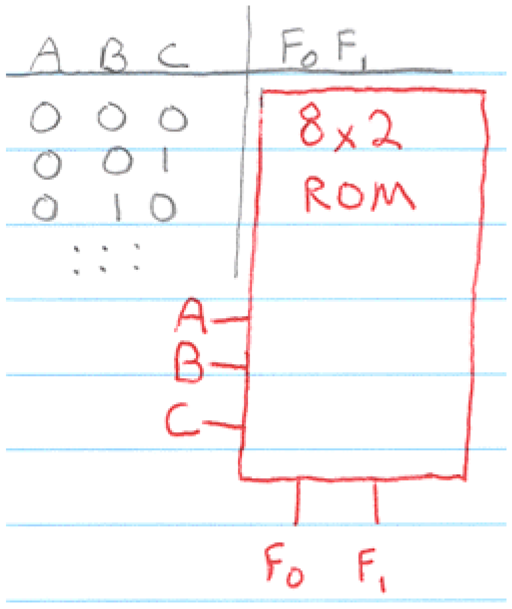
A ROM is a combinational component for storing data. The data might be a truth table or the data might be the control words for a microprogrammed CPU. (This is a topic we will discuss later. Control words in a microprogrammed CPU interpret the macro instructions understood by the CPU.)

A ROM can be programmed at the factory or in the field.

The following image shows the generic form of a ROM:



An n x m ROM can store the truth table for m functions defined on $\log_2 n$ variables:



Example: Implement the following functions in a ROM:

$$F_0 = A$$

$$F_1 = A'B' + AB$$

Since a ROM stores the complete truth table of a function (or you could say that a ROM decodes every minterm of a function) the first step is to express each function as a truth table.

A B	**F_0 F_1**
0 0	0 1
0 1	0 0

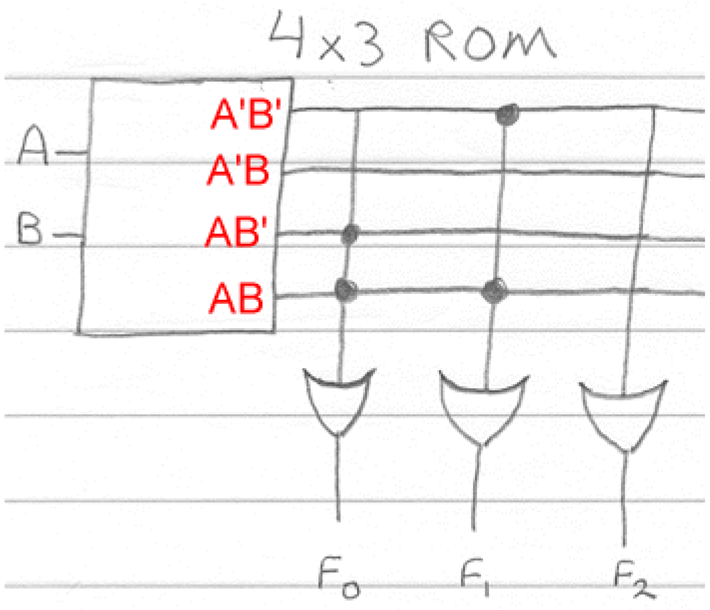
1 0	1 0
1 1	1 1

For the discussion that follows it may be helpful to keep in mind the canonical form of the function also:

$$F_0 = AB' + AB$$

$$F_1 = A'B' + AB$$

We use a special notation to show the ROM implementation of a function:



The image above shows how a 4x3 ROM can be used to implement the two functions F_0 and F_1. (Note, there is room in the ROM for 3 functions of two variables. F_2 isn't used. I'm showing a ROM with an unused portion to demonstrate that a ROM may still be the most efficient implementation even when large sections of the ROM go unused.) You can imagine a decoder inside the ROM that decodes the inputs A B. Just like the decoder we defined earlier, one output line is selected for every unique set of inputs. If the selected output line is connected to an OR gate the function associated with the OR gate will have a value of 1 for the particular set of inputs.

A single vertical line that intersects 4 horizontal lines represents potentially 4 different lines or connections. This is a notational convenience we will use when talking about ROM's, PLA's, and PAL's because it makes the diagrams much easier to read.

The circles at the intersection of two lines indicates a connection. Connections are either formed at the factory or in the field. If they are formed in the field a special programmable

ROM is used. One type of programmable ROM is a ROM that has a fuse at every connection. Fuses at connections not wanted are burned by running high current through the fuse. What is left are the connections that define the data within the ROM.

Observations

- Not very efficient implementation of sparse functions.
- A ROM that implements two functions does not require twice the number of gates as a ROM that implements one function. (The decoder is shared by every output function.)

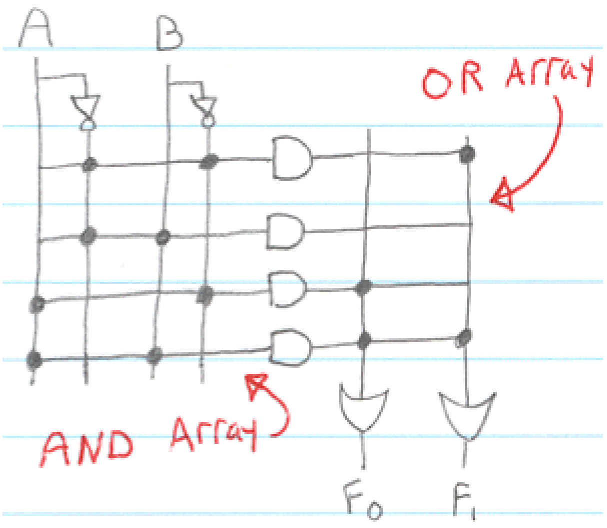
Programmable Logic

A programmable logic device works like a ROM but is a more efficient solution for implementing sparse output functions. (Not all minterms are decoded.)

There are two types of programmable logic devices:

- PLA (Programmable Logic Array)
- PAL (Programmable Array Logic)

We suggested earlier that a ROM had a decoder inside it. You could visualize this as:



The image above also defines two terms we will use to distinguish between PLA and PAL devices:

- AND Array - this is the portion of the device that decodes the inputs. The AND array determines the minterms decoded by the device. A ROM decodes all possible minterms.
- OR Array - this is the portion of the device that combines the minterms for the definition of a function.

PLA

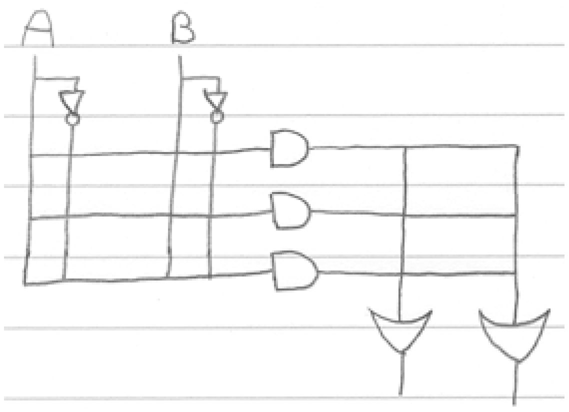
A PLA is a programmable logic device with a programmable AND array and a programmable OR array.

A PLA with n inputs has fewer than 2^n AND gates (otherwise there would be no advantage over a ROM implementation of the same size). A PLA only needs to have enough AND gates to decode as many unique terms as there are in the functions it will implement.

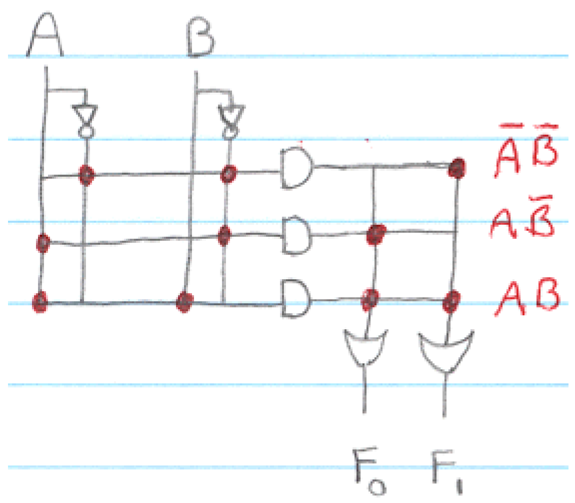
Because we can control the AND array and there is a limit to the number of terms that can be specified in the AND array, it may be more economical to simplify the function before implementing it with a PLA. If you do simplify the function and intend to implement with a PAL device you should also keep in mind that product terms can be shared between functions. (Product sharing is when two functions share a product term decoded by the AND array. For example, in the image below the product term AB is shared between F_0 and F_1.)

Example: Implement the functions F_0 F_1 we introduced above using a PLA with 2 inputs, 3 product terms, and 2 outputs.

The unprogrammed PLA from the manufacture looks like:



After programming for the two functions F_0 F_1 the state of the PLA is:



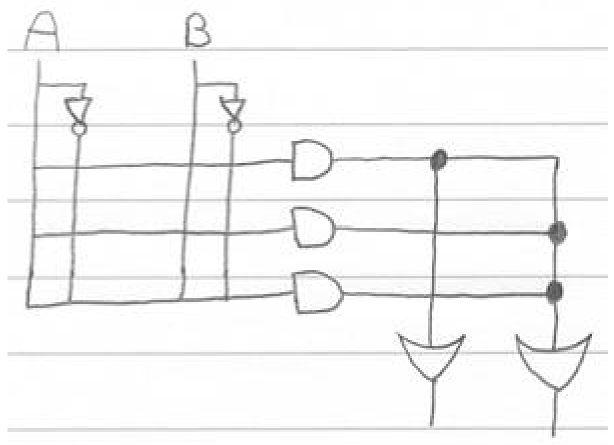
Notice that we only need three AND gates because there are only three unique minterms in the functions F_0 and F_1. Also, notice that since we have control over the OR array we can share the minterm AB in the definitions of both functions.

Note, there may be an advantage to simplifying the functions before implementing. In the example used here there is no advantage. The simplified form of the functions F_0 and F_1 still require 3 unique product terms. Because product terms can be shared between functions its important to look for common product terms when simplifying.

PAL

A PAL is a programmable logic device with a programmable AND array and a fixed OR array.

A PAL has a fixed OR array. For example, here is what an unprogrammed PAL might look like straight from the manufacture:



A fixed OR array makes the device less expensive to manufacture. On the other hand, having a fixed OR array means you can't share product terms between functions.

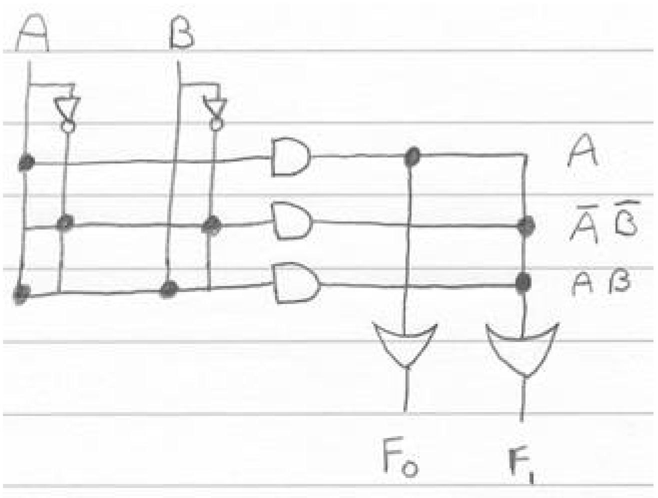
Example: Implement the functions F_0 F_1 we introduced above using the PAL given above.

For this implementation we will need to simplify the functions F_0 F_1 because the PAL we are given has an output function that can accommodate only one product term. The simplified form of the functions are:

$$F_0 = A$$

$$F_1 = A'B' + AB$$

After programming for the two functions F_0 F_1 the state of the PAL is:



So, in summary:

- A PLA device has a programmable AND and programmable OR array
- A PAL device has a programmable AND and fixed OR array
- (You could also say that a ROM has a fixed AND and programmable OR array)

When implementing with a ROM there is no advantage to minimizing the functions since the input is fully decoded. When implementing with a PLA there may be an advantage to minimizing the expression but you also have to keep in mind that product terms can be shared between functions. So, when you are minimizing one function you need to consider the form of other functions and watch for product terms that can be shared. When implementing with a PAL there may also be some advantages to minimizing the function first. However, since you can't share product terms with a PAL you don't have to consider the form of other functions when minimizing.

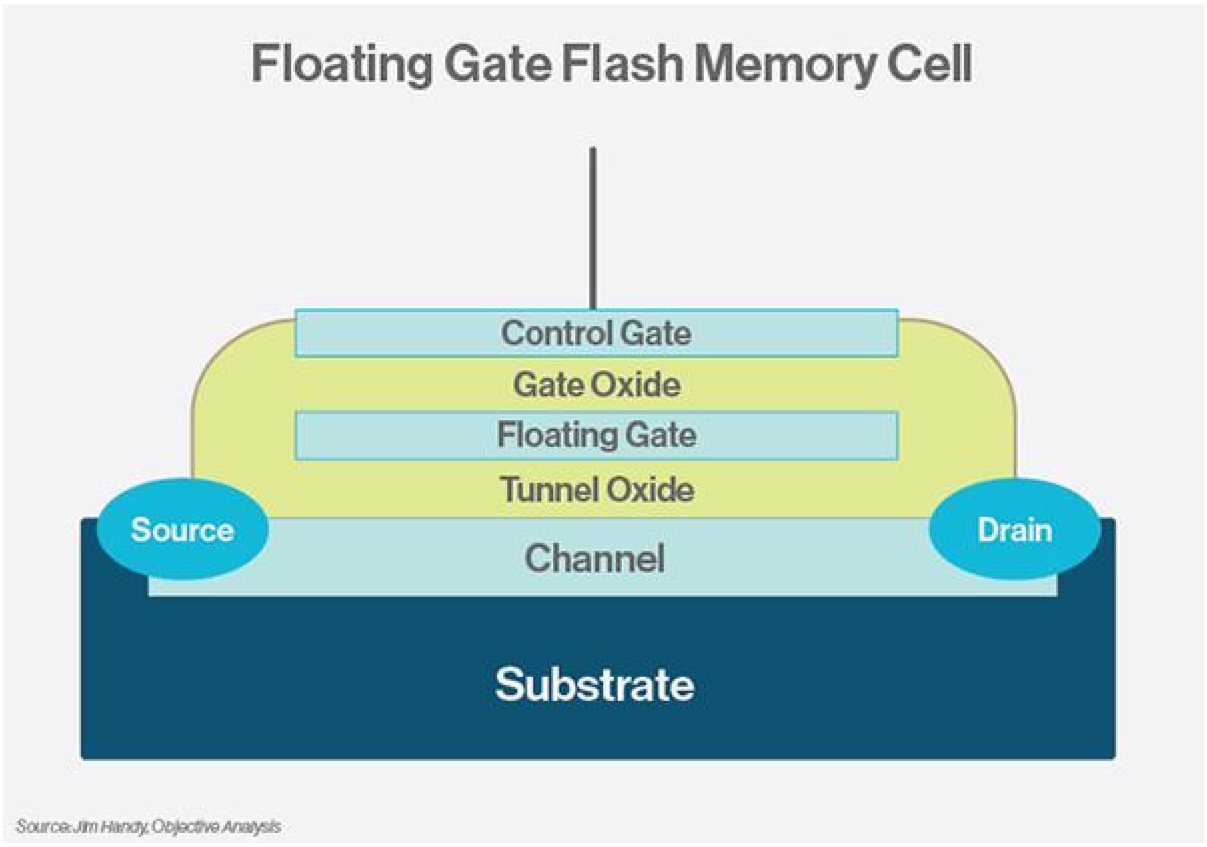
FLASH MEMORY

Dr. Fujio Masuoka is credited with the invention of flash memory when he worked for Toshiba in the 1980s. Masuoka’s colleague, Shoji Ariizumi, coined the term *flash* because the process of erasing all the data from a semiconductor chip reminded him of the flash of a camera.

Flash memory evolved from erasable programmable read-only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM). Flash is technically a variant of EEPROM, but the industry reserves the term *EEPROM* for byte-level erasable memory and applies the term *flash memory* to larger block-level erasable memory. Devices using flash memory erase data at the block level and rewrite data at the byte level (NOR flash) or multiple-byte page level (NAND flash). Flash memory is widely used for storage and data transfer in consumer devices, enterprise systems and industrial applications.

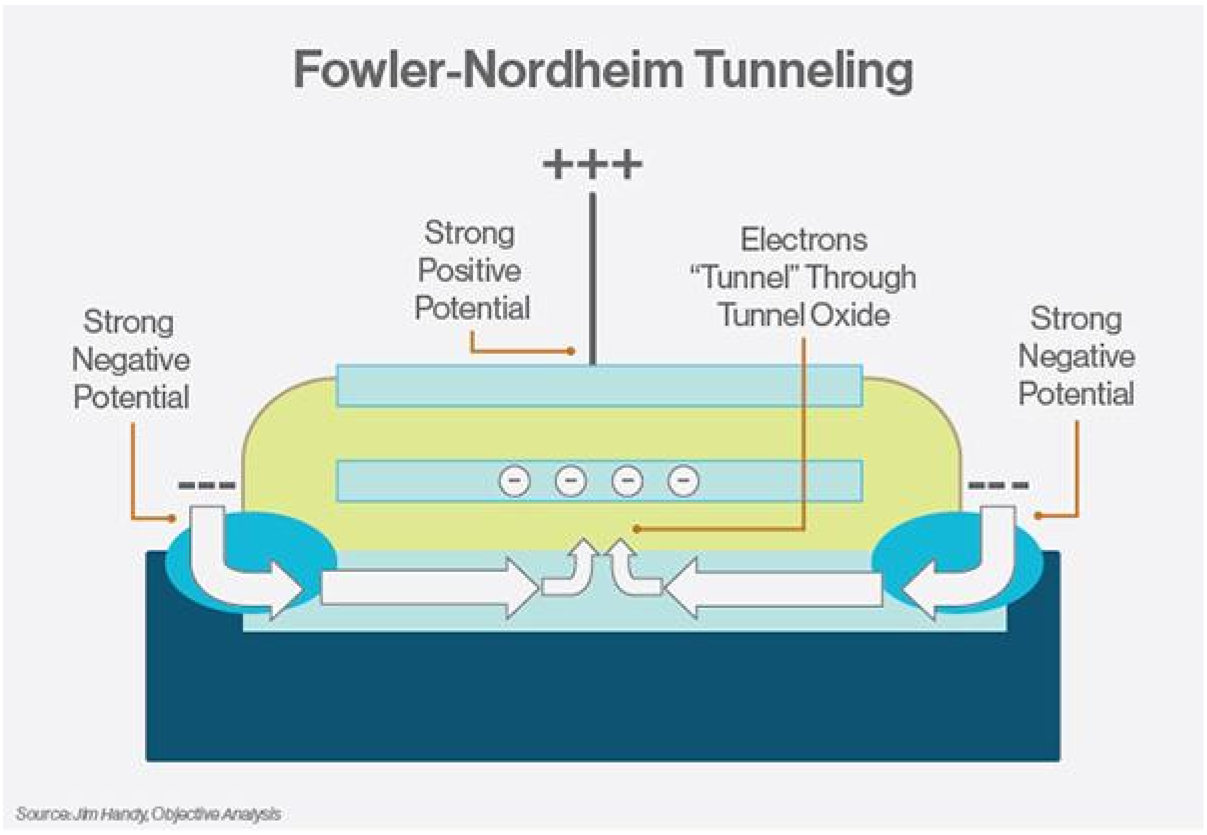
How flash memory works

A basic flash memory cell consists of a storage transistor with a control gate and a floating gate, which is insulated from the rest of the transistor by a thin dielectric material or oxide layer. The floating gate stores the electrical charge and controls the flow of the electrical current.



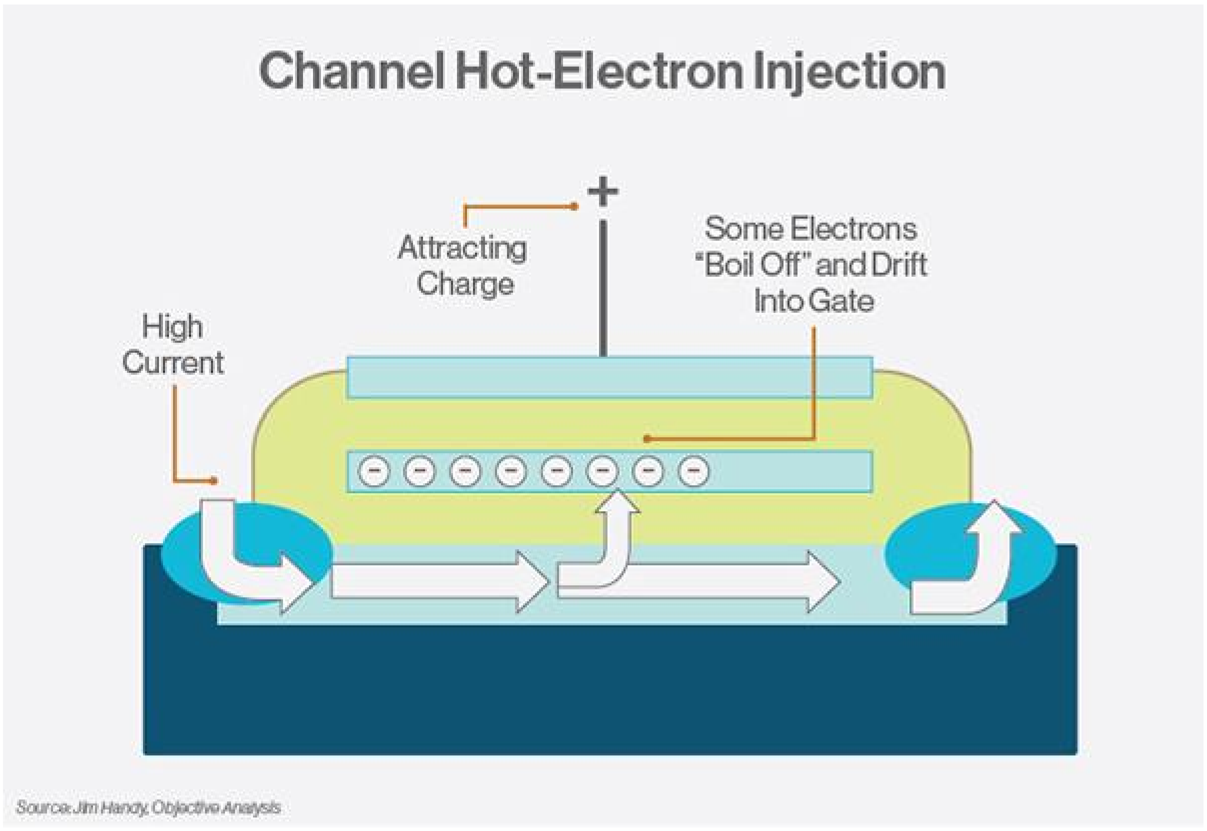
Electrons are added to or removed from the floating gate to change the storage transistor’s threshold voltage to program the cell to be a zero or a one. A process called *Fowler-Nordheim tunneling* removes electrons from the floating gate. Either Fowler-Nordheim tunneling or a phenomenon known as *channel hot-electron injection* traps the electrons in the floating gate.

When erasing through Fowler-Nordheim tunneling, a strong negative charge on the control gate forces electrons off the floating gate and into the channel, where a strong positive charge exists. The reverse happens when using Fowler-Nordheim tunneling to trap electrons in the floating gate. Electrons are able to forge through the thin oxide layer to the floating gate in the presence of a high electric field, with a strong negative charge on the cell's source and the drain and a strong positive charge on the control gate.


Fowler-Nordheim Tunneling
+++
Strong Positive Potential
Electrons "Tunnel" Through Tunnel Oxide
Strong Negative Potential

Strong Negative Potential
Source: Jim Handy, Objective Analysis

With channel hot-electron injection (or hot-carrier injection), electrons gain enough energy from the high current in the channel and attracting charge on the control gate to break through the gate oxide and change the threshold voltage of the floating gate.


Channel Hot-Electron Injection
Attracting Charge
+
Some Electrons "Boil Off" and Drift Into Gate
High Current
Source: Jim Handy, Objective Analysis

Electrons are trapped in the floating gate, whether a device containing the flash memory cell is powered on or off, because of the electrical isolation created by the oxide layer.

EPROM and EEPROM cells operate similarly to flash memory in writing, or programming, data, but they differ from flash memory in the way they erase data. An EPROM is erased by removing the chip from the system and exposing the array to ultraviolet light to erase data. An EEPROM erases data electronically at the byte level, while flash memory erases data electronically at the block level.

NOR vs. NAND flash memory

There are two types of flash memory: NOR and NAND.

NOR and NAND flash memory differ in architecture and design characteristics. NOR flash uses no shared components and can connect individual memory cells in parallel, enabling random access to data. A NAND flash cell is more compact in size, with fewer bit lines, and strings together floating-gate transistors to achieve greater storage density. NAND is better suited to serial rather than random data access.

NOR flash is fast on data reads, but it is typically slower than NAND on erases and writes. NOR flash programs data at the byte level. NAND flash programs data in pages, which are larger than bytes but smaller than blocks. For instance, a page might be 4 kilobytes (KB), while a block might be 128 KB to 256 KB or megabytes in size. NAND flash uses less power than NOR flash for write-intensive applications.

NOR flash is more expensive to produce than NAND flash and tends to be used primarily in consumer and embedded devices for boot purposes and read-only code-storage applications. NAND flash is more suitable for data storage in consumer devices and enterprise server and storage systems due to its lower cost per bit to store data, greater density, and higher programming and erase speeds.

Devices such as a camera phone may use both NOR and NAND flash in addition to other memory technologies to facilitate code execution and data storage.

Pros and cons of flash memory

Flash is the least expensive form of semiconductor memory. Unlike dynamic random access memory (DRAM) and static RAM (SRAM), flash memory is nonvolatile, offers lower power consumption and can be erased in large blocks. Also on the plus side, NOR flash offers fast random reads, while NAND flash is fast with serial reads and writes

A solid-state drive (SSD) with NAND flash memory chips delivers significantly higher performance than traditional magnetic media such as hard disk drives (HDDs) and tape. Flash drives also consume less power and produce less heat than HDDs. Enterprise storage systems equipped with flash drives are capable of low latency, which is measured in microseconds or milliseconds.

The main disadvantages of flash memory are the wear-out mechanism and cell-to-cell interference as the dies get smaller. Bits can fail with excessively high numbers of program/erase cycles, which eventually break down the oxide layer that traps electrons. The deterioration can distort the manufacturer-set threshold value at which a charge is determined to be a zero or a one. Electrons may escape and get stuck in the oxide insulation layer leading to errors.

Anecdotal evidence suggests NAND flash drives are not wearing out to the degree once feared. Flash drive manufacturers have improved endurance and reliability through error correction code algorithms, wear leveling and other technologies. In addition, SSDs do not wear out without warning. They typically alert users in the same way a sensor might indicate an underinflated tire.

NAND flash memory storage types

NAND flash semiconductor manufacturers have developed different types of memory suitable for a wide range of data storage uses cases. NOR flash memory types

The two main types of NOR flash memory are parallel and serial (also known as serial peripheral interface). NOR flash originally was available only with a parallel interface. Parallel NOR offers high performance, security and additional features; its primary uses include industrial, automotive, networking, and telecom systems and equipment. Serial NOR flash has lower pin counts and smaller packaging and is less expensive than parallel NOR. Use cases for serial NOR include personal and ultra-thin computers, servers, HDDs, printers, digital cameras, modems and routers.

Flash memory producers and products

Major manufacturers of NAND flash memory chips include Intel, Micron, Samsung, SanDisk, SK Hynix and Toshiba. Major manufacturers of NOR flash memory include Macronix, Microchip, Micron, Spansion and Winbond. Flash memory is used in enterprise server, storage and networking technology as well as a wide range of consumer devices, including USB drives, mobile phones, digital cameras, tablet computers, PC cards in notebook computers and embedded controllers. For instance, NAND flash-based SSDs are often used to accelerate the performance of I/O-intensive applications. NOR flash memory is often used to hold control code such as the basic input/output system (BIOS) in a PC.Flash memory is seeing growing use for in-memory computing to help speed performance and increase the scalability of systems that manage and analyze enormous amounts of data.

Content-addressable memories (CAMs) are hardware search engines that are much faster than algorithmic approaches for search-intensive applications. CAMs are composed of conventional semiconductor memory (usually SRAM) with added comparison circuitry that enable a search operation to complete in a single clock cycle. The two most common search-intensive tasks that use CAMs are packet forwarding and packet classification in Internet routers.

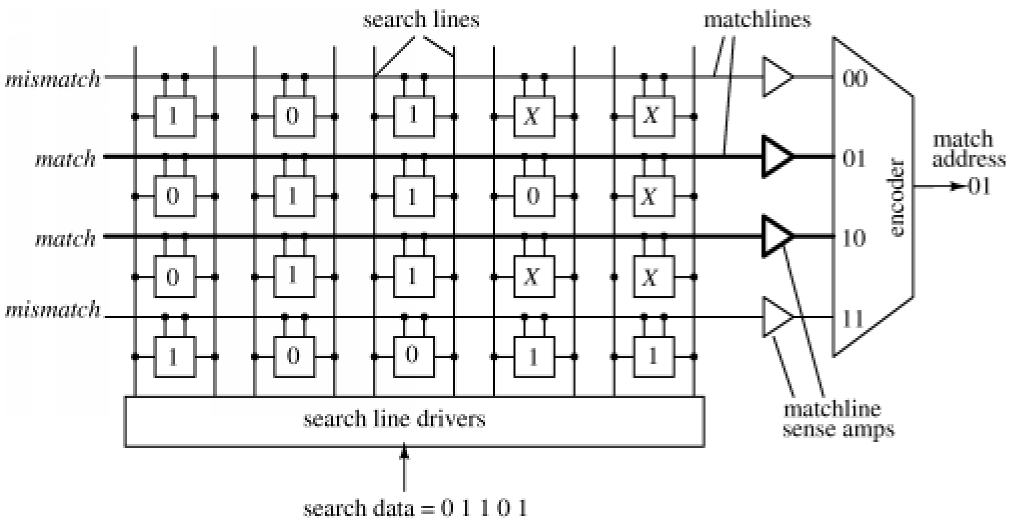
1	101XX	A
2	0110X	B
3	011XX	C
4	10011	D

The routing parameters that determine the complexity of the implementation are the entry size, the table size, the search rate, and the table update rate. Present IPv4 addresses are 32 bits long and proposed IPv6 addresses are 128 bits long. Ancillary information like the source address and quality-of-service (QoS) information can balloon IPv6 routing table entry sizes to 288—576 bits. Currently, routing table sizes are about 30,000 entries but are growing rapidly. Terabit-class routers must perform hundreds of millions of searches per second in addition to thousands of routing table updates per second.

There are many software-based methods to implement the address lookup function [RSBD01], although not all can meet the above requirements. For example, software-based binary searching accomplishes the task if the lookup table is ordered. Binary searching has O(log n) time complexity in addition to the extra time required to insert a new entry in the table. Almost all algorithmic approaches are too slow to keep up with projected routing requirements. In contrast, CAMs use hardware to complete a search in a single cycle, resulting in constant O(1) time complexity. This is accomplished by adding comparison circuitry to every cell of hardware memory. The result is a fast, massively parallel lookup engine. The strength of CAMs over algorithmic approaches is their high search throughput. The current bottleneck is the large power consumption due to the large amount of comparison circuitry activated in parallel. Reducing the power consumption is a key aim of current CAM research.

Line No.	Address (Binary)	Output Port
1	101XX	A
2	0110X	B
3	011XX	C
4	10011	D

There are two basic forms of CAM: binary and ternary. Binary CAMs support storage and searching of binary bits, zero or one (0,1). Ternary CAMs support storing of zero, one, or don't care bit (0,1,X). Ternary CAMs are presently the dominant CAM since longest-prefix routing is the Internet standard. Figure 1 shows a block diagram of a simplified 4 x 5 bit ternary CAM with a NOR-based architecture. The CAM contains the routing table from Table 1 to illustrate how a CAM implements address lookup. The CAM core cells are arranged into four horizontal words, each five bits long. Core cells contain both storage and comparison circuitry. The search lines run vertically in the figure and broadcast the search data to the CAM cells. The matchlines run horizontally across the array and indicate whether the search data matches the row's word. An activated matchline indicates a match and a deactivated matchline indicates a non-match, called a mismatch in the CAM literature. The matchlines are inputs to an encoder that generates the address corresponding to the match location.



A CAM search operation begins with precharging all matchlines high, putting them all temporarily in the match state. Next, the search line drivers broadcast the search data, 01101 in the figure, onto the search lines. Then each CAM core cell compares its stored bit against the bit on its corresponding search lines. Cells with matching data do not affect the matchline but cells with a mismatch pull down the matchline. Cells storing an X operate as if a match has occurred. The aggregate result is that matchlines are pulled down for any word that has at least one mismatch. All other matchlines remain activated (precharged high). In the figure, the two middle matchlines remain activated, indicating a match, while the other matchlines discharge to ground, indicating a mismatch. Last, the encoder generates the search address location of the matching data. In the example, the encoder selects numerically the smallest numbered matchline of the two activated matchlines, generating the match address 01. This match address is used as the input address to a RAM that contains a list of output ports as depicted in Figure 2. This CAM/RAM system is a complete implementation of an address lookup engine. The match address output of the CAM is in fact a pointer used to retrieve associated data from the RAM. In this case the associated data is the output port. The CAM/RAM search can be viewed as a dictionary lookup where the search data is the word to be queried and the RAM contains the word definitions. With this sketch of CAM operation, we now look at the comparison circuitry in the CAM core cells.

CHARGE COUPLED DEVICE (CCD)

Fundamentally, a charge coupled device (CCD) is an integrated circuit etched onto a silicon surface forming light sensitive elements called pixels. Photons incident on this surface generate charge that can be read by electronics and turned into a digital copy of the light patterns falling on the device. CCDs come in a wide variety of sizes and types and are used in many applications from cell phone cameras to high-end scientific applications. The charge-coupled device was invented in 1969 at AT&T Bell Labs by Willard Boyle and George E. Smith.

Basics of operation

An image is projected through a lens onto the capacitor array (the photoactive region), causing each capacitor to accumulate an electric charge proportional to the light intensity at that location. A one-dimensional array, used in line-scan cameras, captures a single slice of the image, whereas a two-dimensional array, used in video and still cameras, captures a two-dimensional picture corresponding to the scene projected onto the focal plane of the sensor. Once the array has been exposed to the image, a control circuit causes each capacitor to transfer its contents to its neighbor (operating as a shift register). The last capacitor in the array dumps its charge into a charge amplifier, which converts the charge into a voltage. By repeating this process, the controlling circuit converts the entire contents of the array in the semiconductor to a sequence of voltages. In a digital device, these voltages are then sampled, digitized, and usually stored in memory; in an analog device (such as an analog video camera), they are processed into a continuous analog signal (e.g. by feeding the output of the charge amplifier into a low-pass filter), which is then processed and fed out to other circuits for transmission, recording, or other processing.

Field-programmable gate arrays (FPGAs)

Field-programmable gate arrays (FPGAs) are reprogrammable silicon chips. Ross Freeman, the cofounder of Xilinx, invented the first FPGA in 1985. FPGA chip adoption across all industries is driven by the fact that FPGAs combine the best parts of application-specific integrated circuits (ASICs) and processor-based systems. FPGAs provide hardware-timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design.

Reprogrammable silicon also has the same flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing.

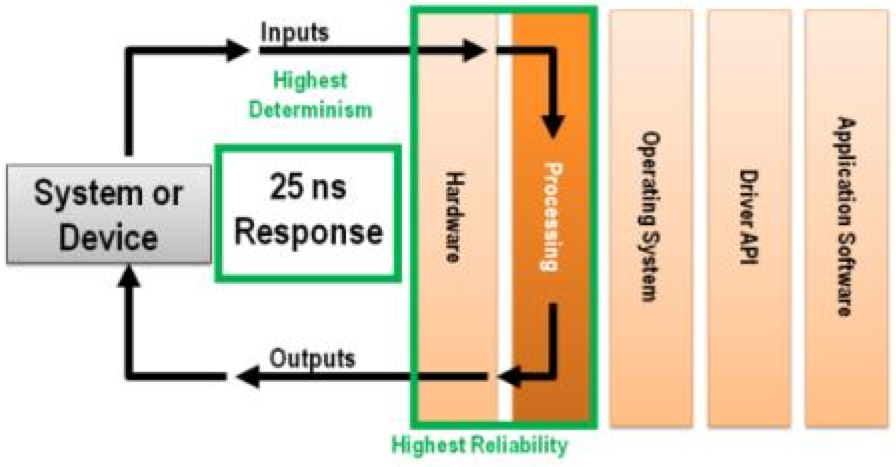


Figure 1.One of the benefits of FPGAs over processor-based systems is that the application logic is implemented in hardware circuits rather than executing on top of an OS, drivers, and application software.

Defining the Parts of an FPGA

Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit and I/O blocks to allow the circuit to access the outside world.

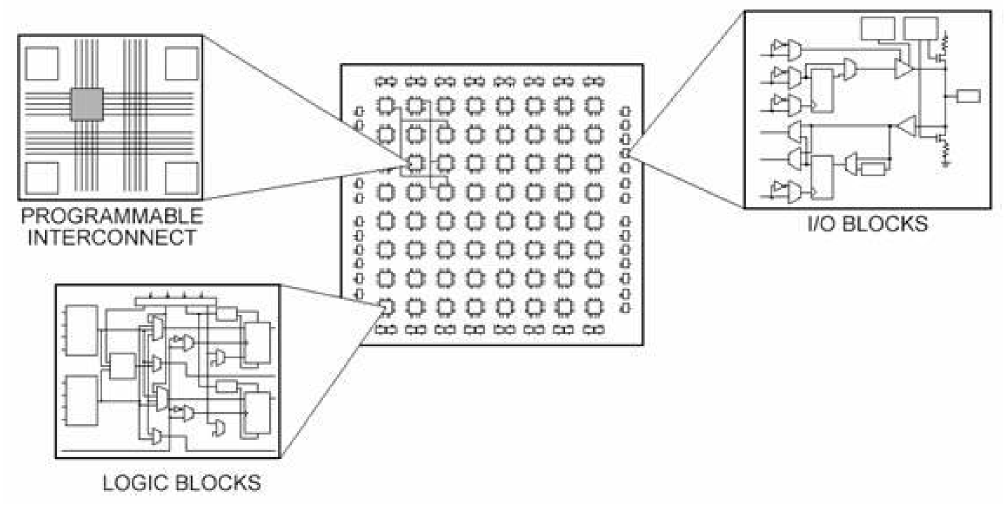


Figure 2. The Different Parts of an FPGA

FPGA resource specifications often include the number of configurable logic blocks, number of fixed function logic blocks such as multipliers, and size of memory resources like embedded block RAM. Of the many FPGA chip parts, these are typically the most important when selecting and comparing FPGAs for a particular application.

The configurable logic blocks (CLBs) are the basic logic unit of an FPGA. Sometimes referred to as slices or logic cells, CLBs are made up of two basic components: flip-flops and lookup tables (LUTs). Various FPGA families differ in the way flip-flops and LUTs are packaged together, so it is important to understand flip-flops and LUTs.